Lab Project 16.2: Solid Leaf BSP Compiler

As we move forward with the development of our graphics engine and framework, certain processes will become much more expensive for us to perform. For instance, many of the techniques employed in providing accurate scene visibility and occlusion culling schemes can involve extremely lengthy computations which cannot possibly be done in real time. In order to prevent these computations from introducing a significant delay, which could destroy the gaming experience for the player, it is important that load times remain as brief as possible. As a result, it is very often necessary to find ways to remove or optimize any processes or calculations which may need to be performed while a level is being loaded. Even in common situations, such as the compilation of a relatively simple oct-tree or kD-tree, these additional computations can easily add several seconds or more to a level's loading time. This is of course on top of the many other load-time tasks that have to be undertaken such as the reading of files, loading and potential compressing of textures, building of animation data and so on. For this reason, it will become of much greater importance that we begin to remove as much of the burden of computing such complex information from the runtime portion of our application.

Throughout the previous chapters we have introduced several new concepts and techniques which can introduce a significant delay in load time execution if we were to use a scene with a high level of geometric complexity. Constructing a spatial hierarchy from our level is by no means a trivial task as we have already discovered. In the next chapter we will be introducing the foundation of our visibility determination scheme. The operations involved in such a scheme can often take anywhere from several minutes to many *hours* to execute. To perform such lengthy operations at runtime is, as a result, unfeasible at best. One method by which we can reduce the time that it takes to load and construct our level data is to try and move as many processes and calculations as is possible from our application altogether.

As we know, most everything that we have implemented in these spatial partitioning lessons is intended to build information that allows our application to more easily manage scene data. With the exception of those processes which rely on information only available on the end user's computer hardware – such as how much data can be stored in a single vertex buffer for instance – most of the data that is constructed is based on information that does not change. As an example, during the construction of an oct-tree, kD-tree or quad-tree, the input geometry is used to define how the tree is constructed. This means that we can fully expect identical tree structures to be built on any computer system as long as they each use the same input geometry and construction parameters. Furthermore, each of these tree building classes is fundamentally assembling a series of simple data structures that are used by the application in some manner after the main game loop has commenced.

So, we know that the structure of the tree will not change irrespective of the end-user system on which it is built. As a result, it makes sense that this same information can just as easily and accurately be constructed on the developer or artist's hardware. We also know that once the game play portion of our application has begun we are simply making use of information that was built during the compilation stage. It should be possible therefore to simplify this problem by having the developer or artist save this information out to a file alongside the scene itself. This reduces our application's involvement in this process to one of simply loading and reconstructing the original data structures, rather than having to compile it on the fly. This type of operation is often referred to as a pre-process. Three such operations that exemplify such pre-process tasks – and those that will be discussed in this chapter's workbook – are

Hidden Surface Removal, *Spatial Partitioning* (of any kind) and *T-Junction Repair*. Due to the focused nature of these individual compilation tasks, it is often desirable to build a tool that is completely separate from our game application and whose sole purpose is to perform these potentially lengthy processes in advance of runtime.

With lab project 16.2, we will begin developing such a tool that will perform the brunt of our scene processing and compilation tasks. By using a relatively modular design, we will also be able to provide a level of extendibility that will be to our great benefit as we introduce additional pre-process tasks in the future.

In this section will cover the following:

- The concepts involved in building a reusable tool.
- A series of new math utility classes that will be used by this application.
- Adapting our application structures for use in a file based system.
- Building a portable pre-processing core.
- Designing a modular system for the compilation processes.
- Compilation of a polygon-aligned BSP leaf tree with solid / empty leaf information.
- Implementation of the hidden surface removal process.
- Porting the T-Junction repair system to the pre-processing tool.

Building a Reusable Tool

One of the first things that should be apparent when initially examining the source code to our new solid leaf BSP compilation tool is how different the compiler portion may appear when compared to everything we have encountered in our lab projects to date. One such difference is the implementation of several new math support classes that have been introduced in order to remove a specific reliance on the third party D3DX library provided by the DirectX™ SDK. Whilst this may seem a little superfluous to begin with, it is important to bear in mind that when we are developing a generic tool such as this we will often want to ensure that it is as portable and **re-usable** as possible.

In contrast to our game application – which is generally targeted at a specific platform / API combination – the broad applicability of the types of compilation process implemented in such tools make their re-usability highly desirable. By developing our pre-processing application in such a way that it has as few unnecessary API and platform dependencies as possible will allow this same tool to be adapted and applied to almost any project you may undertake in the future, regardless of the target platform or API. If you plan to license your toolset to third party developers, this can also be of great importance to them.

One other important benefit gained by developing our compilation toolset in this way is that it greatly increases its longevity. This is due to the fact that it will remain unaffected by any future changes made to any API on which it is dependant. Finally, by substituting the game-centric D3DX library with that of our own design we will be able to write code in a manner which best suits the specific application we are building.

The Replacement Math Classes

As a direct result of removing the dependency on the DirectX™ and associated APIs, we have introduced five new support classes which provide the math library functionality that will be required when building our compilation toolset. These include a plane class similar to D3DXPLANE, a matrix class which replaces D3DXMATRIX, two vector classes analogous to D3DXVECTOR3 and D3DXVECTOR4, and finally a new and more easily manageable axis aligned bounding box class. Each of these classes exposes member functions which provide the functionality previously accessed via calls to global D3DX functions such as D3DXMatrixMultiply in addition to other common functions that will be required.

The following list outlines each of the new math classes we will be introducing in the new compilation toolset. We will use the ‘CVector3’ class as the means by which we discuss the general design concepts used throughout our new supporting math tools. As a result we will discuss this class in greater detail than those which follow. In each case however, we will provide a summary of that class along with its intended purpose.

- **CVector3**

  This new class is designed to replace the D3DXVECTOR3 structure and associated support routines that we have been using within our run-time lab projects to date. There are many uses to

which this has been put in the past and all of the same D3DX functionality has been included here in the form of member functions. As an example of this, when performing a dot product between two D3DX vector types, we have previously made use of the 'D3DXVec3Dot' global function. With this new vector class, the dot product function has been included within the class itself via the 'CVector3::Dot' member.

In our previous implementations we have performed a dot product operation between two vectors using code similar to that shown below:

```
D3DXVECTOR3 Vector1( 1.0f, 0.0f, 0.0f );
D3DXVECTOR3 Vector2( 0.0f, 1.0f, 0.0f );

// Perform the dot product
float fDot = D3DXVec3Dot( &Vector1, &Vector2 );
```

This code should be very familiar to you at this point. As you can see we have simply built two vector classes on the stack, specifying some initial values to the class constructor. Once we have created these two vector objects, we simply pass the pointers to both of these into the 'D3DXVec3Dot' function at which point the result is returned. As mentioned previously, our new vector support class implements this same functionality using a member function rather than a global one. As a result, the code we would use in order to perform a dot product on these same two vectors now looks similar to the following:

```
CVector3 Vector1( 1.0f, 0.0f, 0.0f );
CVector3 Vector2( 0.0f, 1.0f, 0.0f );

// Perform the dot product
float fDot = Vector1.Dot( Vector2 );
```

Although the differences are relatively minor, one important thing to notice is how we have called the dot product member of 'Vector1' and simply passed 'Vector2' in as a single parameter. As before however, we are still requesting that the dot product function calculates the resulting value between the same two vectors – 'Vector1' and 'Vector2'. By using member functions in this way, we can often simplify the code that we would need to otherwise write. A good example of such a case is when we might need the ability to perform vector operations on data which is not explicitly a D3DXVECTOR3 object. Such cases have often arisen in the past when we must use *vertex* data as the input to the global vector functions. In the past, it has often been necessary to perform a complex series of casts and de-references in order to achieve this:

```
CVertex     Vertices[3];
D3DXVECTOR3 Edge1, Edge2, Normal ;

// Define Vertices
Vertex[0] = CVertex( 10.0f, 10.0f, 0.0f );
Vertex[1] = CVertex( 10.0f, 20.0f, 0.0f );
Vertex[2] = CVertex( 20.0f, 10.0f, 0.0f );

// Calculate triangle normal
D3DXVec3Normalize(&Edge1,
                  &((D3DXVECTOR3&)Vertices[1] - (D3DXVECTOR3&)Vertices[0]));
D3DXVec3Normalize(&Edge2,
                  &((D3DXVECTOR3&)Vertices[2] - (D3DXVECTOR3&)Vertices[0]));

// Cross the two edges
D3DXVec3Cross( &Normal, &Edge1, &Edge2 );

// Normalize the result
D3DXVec3Normalize( &Normal, &Normal );
```

As the complexity and length of the required operation increases, it can often become difficult to follow the indented outcome of that code. One of the primary advantages in creating a vector class in this way is that we are able to derive other classes from it. This could include our *vertex* class for instance. If we were to derive our vertex class from the new 'CVector3' class, we will gain the ability to perform vector like operations using the derived 'CVertex' object directly:

```
CVertex  Vertices[3];
CVector3 Edge1, Edge2, Normal;

// Define Vertices
Vertex[0] = CVertex( 10.0f, 10.0f, 0.0f );
Vertex[1] = CVertex( 10.0f, 20.0f, 0.0f );
Vertex[2] = CVertex( 20.0f, 10.0f, 0.0f );

// Calculate triangle normal
Edge1  = (Vertices[1] – Vertices[0]).Normalize();
Edge2  = (Vertices[2] – Vertices[0]).Normalize();
Normal = Edge1.Cross( Edge2 ).Normalize();
```

Another item of note is that the members defined within our new vector class no longer require that we pass the input parameters as pointers. Instead we have chosen to use parameter references in most of the class member functions. Recall that references are essentially the same as pointers in that they specify the memory address to an underlying data item. This provides us with most of the benefits and efficiency we would gain by using pointers but without the need to use the address-of (&) operator with every variable we use as an input parameter. This also helps us to create code which is simple and relatively easy to read and follow.

Finally, notice the method by which we have normalized the result of the cross product operation in the above code. By implementing this functionality using members within the vector class, we can often perform many operations in one go as demonstrated. This can reduce the amount of

individual steps necessary to implement such a procedure in code and can often remove the necessity that we keep temporary storage variables at our disposal for transferring the result from one step into the next. At its core, the C++ compiler is of course still creating code which allocates a temporary variable in order to achieve this. However, in this case we will arguably be able to write cleaner code, in a shorter amount of time. In addition, because we have removed the reliance on pointers and can reduce the overall amount of variables we need to keep track of, our code will often contain fewer errors resulting from incorrect casting and pointer operations or the misuse or misplacement of a temporary variable. As you might imagine, the overall complexity of the compilation tasks we will be implementing will increase dramatically as we move forward throughout this lesson and the next. This places a large degree of importance – from a coding perspective – in keeping the commonly used core libraries as clean and simple to use as possible. Hopefully, by doing so, we will eliminate many of the problems we might need to track down and debug as a result.

In the case of a game application, speed of execution is obviously paramount. To a certain degree during the development of a pre-processing tool, we do have the added advantage of being able to balance run-time execution speed against the ease with which we can develop our application. It could be argued that the compilation tasks our application must undertake may execute more slowly by not using the highly tuned D3DX mathematics library. However, this is not as crucial a concern with such one time pre-process computations as we will be developing.

Before we move on to the next new class, let us finish up by examining a selection of the functions exposed in this new implementation.

Member Functions

As we have already discussed, this new class exposes a series of member functions that provide the familiar D3DX functionality such as the dot product. The following table outlines a few of the most common functions. For a complete list of those functions defined by this class you should take a look at the ‘CVector3’ class declaration in the file named ‘CVector.h’.

Member Function	**D3DX Equivalent**
Dot	D3DXVec3Dot
Cross	D3DXVec3Cross
Length	D3DXVec3Length
SquareLength	D3DXVec3LengthSq
Normalize	D3DXVec3Normalize
TransformCoord	D3DXVec3TransformCoord
TransformNormal	D3DXVec3TransformNormal

Although this is not a complete list of the member functions implemented within the CVector3 class, this table does outline a good selection of the most commonly used D3DX equivalents. Each of these functions works in a manner similar to their D3DX counterparts with the exception

that in many cases, the result is passed back via the return value rather than by setting the contents of a variable passed into the function as a parameter.

In addition to the common vector functionality, there are several additional functions which we have developed in previous lab projects that have also been included here. As before, a sample of these functions is shown in the table below.

Member Function	**Description**
DistanceToPlane	There are many cases in the past where we have needed to calculate the closest distance from a plane at which a point exists. This member wraps this common functionality into a simple function that accepts the required plane as its single parameter.
DistanceToLine	In our previous implementation of the T-Junction repair process, it was necessary to determine the distance between a particular point in space and an edge / line that formed the exterior of a neighbouring polygon. This function wraps this test allowing us simply to pass in the start and end points of such a line and have it return the distance between the vector with which this member is being called and that specified line.
FuzzyCompare	Due to the nature of floating point calculations, it is never a good idea to test whether two vectors describe the same position (or direction) in space by testing for an *exact* equality between their components. We have seen that it is often necessary to use 'Epsilon' or 'Fuzzy' testing to ensure that these inaccuracies do not result in an incorrect comparison. This utility function can be used in cases where we want to test two vectors for equality, given some small tolerance value.

Again, this is not a complete list. However, this small sample should demonstrate the fact that although we have introduced several new concepts into this lab project, it is really just a rearrangement of the concepts and functions we have already developed. In doing so we make the job of developing this pre-processing tool a little more convenient. It also means that we do not have to find creative ways of integrating our rendering and *game specific* classes and utility functions into a framework with a completely different purpose.

Operator Overloads

Just as with the D3DX classes, we have also overridden several of the mathematical and logical operators common to the C++ language. We have often needed to perform mathematical operations directly on our vector structures in the past. An example of such a case is in the calculating of a direction vector by subtracting one positional vector from another. Rather than subtract each of the X, Y and Z components of these vectors individually, it has always been convenient simply to perform this operation on the vectors directly. There are other cases whereby we may want to multiply a vector by a single floating point scalar value for instance. In

this case, overloading the multiplication operator is an ideal candidate because it enables us to multiply our vector *object* by a single floating point value – something which would not otherwise be possible. The table below outlines some of the common operators that have been overloaded in this vector implementation. Again remember that this is not a full list of such operators and you should check out the class declaration for a complete list.

Operator Declaration	**Description**
`CVector3 operator+ ( CVector3& )`	This addition operator accepts a right-hand value of another vector. The implementation for this operator simply adds together each of the respective X, Y and Z components from both the left and right hand values and returns the result as a new vector object. This class also overloads the '+=' operator which functions in a similar manner. However, the right hand vector values are added directly to those of the left hand vector rather than returning a new result.
`CVector3 operator- ( CVector3& )`	Similar to the addition operator overload, this class also overloads the subtraction operator. As before we also overload the '-=' operator to allow a subtraction operation to be directly performed on an existing vector object.
`CVector3 operator* ( float& )`	This multiplication operator accepts a single floating point value as the right-hand argument. This can be used to scale a vector by multiplying each of the left-hand vector's X, Y and Z components by this single scalar value. As before, the '*=' operator for this value type is also overloaded.
`CVector3 operator/ ( float& )`	As with addition and subtraction, we also provide the complement to the multiplication operator in the form of the division operator. This accepts the same floating point argument type and, as with multiplication, simply divides the vector components by this scalar value. As with each of the previous operators, the '/=' operator has been overloaded for this same floating point argument type.

`CVector3 operator* ( CMatrix4& )`	Although we have not yet discussed our new matrix implementation, this vector class provides another multiplication operator overload that accepts a matrix as its right-hand value and returns a three dimensional output vector. As we know, there are two vector-matrix transformation functions that we have at our disposal – TransformCoord and TransformNormal. It is not possible for us to provide two operator overloads that accept the same argument types, so in the case of our vector implementation we have chosen to use the 'TransformCoord' method. This decision was based on the fact that it is often the most commonly used of the two. In the case of 'TransformNormal' however, we can still call the member function directly as we would with D3DX.
`bool operator== ( CVector3& )`	The equality operator is provided in order for us to quickly compare two vector values in a manner similar to that we would use when comparing two integer values. Unlike the aforementioned 'FuzzyCompare' function however, this operator tests for **exact** equality with no tolerance. We did this because it is a more accurate representation of what we expect from the equality operator in general. Whether you choose to use a fuzzy test in this case however, is really a matter of preference and/or convenience.
`bool operator!= ( CVector3& )`	Likewise, the inequality operator is also overloaded to allow us to test for the cases where two vector input values are *not* equal. As with the equality operator, this implementation does not use a fuzzy compare for the inequality test.

Hopefully you can see that our implementation of the 'CVector3' class is very similar to the vector concepts we are already very familiar with. We have tried to keep as close as possible to the D3DX implementation and as a result it should be a relatively simple task to adjust any existing code to use this independent vector class. In addition, by opening up the classes themselves we are able to extend them in a way that would benefit our application as we develop the modules that will rely on this functionality.

Of course, the vector class is not the only component within the D3DX mathematics library that we have come to rely heavily upon. As was mentioned at the beginning of this section, the general design of our new classes will only be discussed in relation to the 'CVector3' class. As a result we will not go into too great a detail about the layout of the remaining classes outlined

over the following pages. So without further ado, let us move on to the next of our five new support classes.

- **CVector4**

The 'CVector4' class is a replacement for 'D3DXVECTOR4' and its associated global functions. This class extends the three dimensional vector previously discussed by including the additional 'w' component as we already know. Although this particular vector class is not used directly by the application we are developing in this lesson, there are several functions available within our math library which accept a four dimensional vector as either an input or output value. An example of such a function is 'D3DXVec3Transform' ('CVector3::Transform' in our new implementation) that takes a three dimensional vector, transforms it by a 4x4 matrix and returns the resulting vector including the w component. As a result, this class has been included to allow these functions to be supported in the event that they may be required for use later in the development cycle.

Similar to its three dimensional counterpart, this class implements much of the four dimensional vector functionality exposed by D3DX within a series of member functions that work directly with the vector data. A sample of the most commonly used functions is again included below, but as before refer to the class declaration in 'CVector.h' for a complete list of those supported.

Member Function	**D3DX Equivalent**
Dot	D3DXVec4Dot
Cross	D3DXVec4Cross
Length	D3DXVec4Length
SquareLength	D3DXVec4LengthSq
Normalize	D3DXVec4Normalize
Transform	D3DXVec4Transform

Again, we are not making use of this class directly within our application at this point. However, we have also provided any appropriate operator overloads in a similar fashion to those discussed in the previous 'CVector3' coverage.

- **CMatrix4**

The 4x4 matrix is probably one of the most commonly used mathematical constructs within game development, second only to that of the three dimensional vector. It is usually quite rare for us to rely heavily upon the use of matrices during the compilation of static scene data. Although this is also true of the compilation techniques we will be implementing in this lesson, there are objects within our scene whose position and orientation are described by matrices. An example of such an object is the game 'entity' which, as we know, is essentially just a generic data storage concept providing drop in scene support for our animating objects, lights, trees etc. Although this external compilation tool will perform no rendering of the scene, this entity

information must still be processed in order for us to transfer the data into what will ultimately be exported as the compiled scene file.

Due to the complex nature of the operations that any matrix implementation is tasked with performing, it is important that the functionality offered by any such matrix class is both efficient and easy to use. Because of the fact that we are already extremely familiar with the 'D3DXMATRIX' class exposed by the D3DX math API, our matrix implementation is based heavily on this class and its associated utility functions. As with the 'CVector3' implementation of the 'D3DXVECTOR3' class, we employ a similar design in which those utility functions are provided as member functions of the matrix class itself.

Member Functions

Some of the D3DX equivalent functions that will most commonly be used are listed in the table below. In the case of our matrix class, it is recommended that you take a look at the class declaration for 'CMatrix4' included in the file named 'CMatrix.h' for a full list of member functions and their parameter layout.

Member Function(s)	**D3DX Equivalent**
Identity	D3DXMatrixIdentity
GetInverse & Invert	D3DXMatrixInverse
Transpose	D3DXMatrixTranspose
RotateAxis	D3DXMatrixRotationAxis
RotateX	D3DXMatrixRotationX
RotateY	D3DXMatrixRotationY
RotateZ	D3DXMatrixRotationZ
Rotate	D3DXMatrixRotationYawPitchRoll
Scale	D3DXMatrixScaling
Translate	D3DXMatrixTranslation
SetPerspectiveFovLH	D3DXMatrixPerspectiveFovLH
SetLookAtLH	D3DXMatrixLookAtLH

One thing that can very often be frustrating about the D3DX matrix API is the fact that almost all of the matrix functions return absolute matrices that will very often be used simply in a further concatenation with an already existing matrix. Take as an example the 'D3DXMatrixRotationAxis' function. As we know, when we examine the matrix that this function generates, we should see a set of values that describes a rotation relative to what is essentially the world axis. In most circumstances we would normally use this result in a further operation, perhaps by concatenating it with an existing matrix in order to rotate an object's world matrix by the values originally passed as parameters to the rotation function. This design can often draw out even a simple operation into a series of matrix creation and multiplication

operations where we essentially just want to rotate an existing matrix. For this reason, many of the new member functions outlined in the previous table include an additional Boolean parameter. This switch allows us to specify whether or not we would like to perform that operation on the matrix itself, rather than simply setting its values such that it *describes* the operation in question. The default in this case is to perform the operation being requested (i.e. to rotate the matrix) but this can be overridden such that the matrix is simply set in a similar fashion to the D3DX functions.

What this slightly altered design allows us to do is to request that a matrix is rotated in a single call – essentially performing the matrix multiplication on our behalf – rather than simply generating a rotation matrix that we will then have to concatenate with our target matrix manually. The code below demonstrates the code we would need to use in order to rotate an existing matrix by 90 degrees about an arbitrary axis using the D3DX library:

```
D3DXMATRIX mtxRotate;

// Generate rotation matrix
D3DXMatrixRotationAxis( &mtxRotate,
                        &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ),
                        1.5707f );

// Concatenate with existing matrix
D3DXMatrixMultiply( &mtxWorld, &mtxWorld, &mtxRotate );
```

In contrast, the following code outlines the same process as applied to our new matrix class.

```
// New method for rotating an existing matrix
mtxWorld.RotateAxis( CVector3( 0.0f, 1.0f, 0.0f ), 1.5707f );
```

This same concept as applied to each of the rotation functions is also carried forward to the 'Scale' and 'Translate' member functions. In a similar way, we can have our existing matrix physically scaled or translated, rather than simply retrieving a scaling or translation matrix for use in an additional multiplication step.

Note: While this technique can be applied to great effect, it is entirely optional. You can override this default behavior by specifying 'true' to the final 'Reset' parameter in each of these functions. This will essentially request that the matrix with which the operation is being performed is reset before calculating the result.

As with our vector implementation, there are some additional utility functions that we have integrated into our matrix class which provide quick and easy solutions to common implementation tasks.

Member Function	Description
IsIdentity	This function examines the contents of the matrix in order to determine whether or not the values describe an identity matrix. If this is found to be the case, this function returns a result of true. This can be especially useful in determining whether or not we need to use this matrix in an operation such as a recursive concatenation step or in a call to 'SetTransform' during rendering. Since the identity test itself is much less expensive than either of these operations, it can often be a good idea to test the state of a matrix using this function before it is considered.
GetIdentity	This is a **static** function that can be used to retrieve a matrix whose values have already been set to the identity state. At many points throughout the development of our applications it has often been necessary to reset – for instance – the current rendering device matrix to identity before rendering certain other objects. Instead of constructing a temporary matrix and setting its values to identity, we can simply call the 'CMatrix::GetIdentity()' function directly, passing the result into the appropriate 'SetTransform' function.
Zero	This function simply clears all internal values, setting each element in the matrix to zero.

Operator Overloads

One thing that was missing from our list of commonly used matrix member functions and their D3DX equivalents was the 'D3DXMatrixMultiply' function. Rather than include this as a class member, we have integrated this functionality by overloading the multiplication operator. Unlike the vector's 'TransformCoord' and 'TransformNormal' functions, the result expected from the matrix multiply operation is unambiguous. As a result, we simply need to provide one multiplication operator overload to add support for matrix multiplication / concatenation. The most obvious benefit of implementing the matrix multiplication operation in this way is that we can concatenate matrices very easily simply by multiplying one instance of the 'CMatrix4' class by another in precisely the same way as we would with any numeric data type as shown below:

```
mtxResult = mtxWorld * mtxTranslation;
```

Whilst there are many other operator overloads provided within the 'CMatrix4' implementation, this is by far the most important. Some of the other operators which have been overloaded include those for matrix addition and subtraction, matrix equality and inequality as well as scalar multiplication and division. For a complete list of all operator overloads provided by this class, refer to the 'CMatrix4' declaration in the project file 'CMatrix.h'.

- **CPlane3**

The plane class is one that we have focused heavily upon in recent lessons. Used throughout each of our spatial partitioning compilation and rendering procedures this is a very important class for us to implement in this processing application. In the past, we have used the D3DXPLANE structure extensively and have become familiar with it. This structure represents a plane using a 'normal' vector and a distance value. The X, Y and Z components of the plane normal in D3DXPLANE are stored in separate member variables named 'a', 'b' and 'c' in keeping with the standard plane equation. The final 'd' value describes the distance to the plane, from the origin, along the reverse of that plane's normal.

In previous lab projects we were often required to cast the plane structure to a 'D3DXVECTOR3' pointer in order to make use of some of the other D3DX global functions designed to work with vector data. In our plane implementation we are representing this information in a similar format, with the exception that the plane normal is stored in a member variable of the type 'CVector3' rather than individual floating point variables. In doing so, we allow our application to perform standard vector operations using the normal of the plane without any type of casting, extraction or manipulation of the plane data in advance. This should help simplify any operations we perform using the plane class during the development of our pre-processing toolset.

Although there are few functions available in D3DX for working with the D3DXPLANE structure, there are several plane related utility functions to which we have already been exposed.

Member Functions

During our coverage of collision detection and spatial partitioning we introduced a series of utility functions designed to classify various different primitive types against a plane. These included classification tests using a polygon, a point and even a ray. In addition, we introduced a plane technique that allowed us to detect any intersection that occurred between a ray and a plane, as well as retrieving the point at which that ray intersected. As we know, each of these tests and classification routines were used in many places throughout our previous spatial partitioning implementations. As we progress to the development of our new solid leaf BSP tree compiler, it will become apparent that this functionality is crucial to the whole compilation process. As a result of this, our plane class exposes several member utility functions that we will need at some point in the very near future:

Member Function	Description
GetPointOnPlane	As you should hopefully be aware at this point, there are two main ways in which a plane can be represented. Obviously we can represent a plane as a normal vector and a distance value, but we can also represent this as a normal and a point that lies somewhere on that plane. In order to provide a level of compatibility between these two methods, this function is provided to allow you to retrieve a point that lies on the plane described by that instance of the class.

ClassifyPoly	During the construction of the oct-tree, kD-tree, quad-tree and BSP tree we have seen how it is necessary to classify our polygons against the separating planes assigned at each node. This is used to determine the child into which certain polygons need to be passed. This function is analogous to the 'CCollision::PolyClassifyPlane' function that we have previously implemented. It returns an enumerator result informing the calling function about the position of the specified polygon in relation to the plane.
ClassifyPoint	Similar to the 'ClassifyPoly' routine, this function is intended to inform the calling function about the position of a point in space in relation to the plane itself. In previous lab projects, this function was found in the CCollision class under the name 'PointClassifyPlane'.
ClassifyRay	As with the other two functions mentioned, this is yet another primitive classification routine that we have already encountered in the 'CCollision::RayClassifyPlane' function. Once again, this function classifies the specified ray against the plane and returns a result depending on whether that ray was in-front, behind, spanning or on-plane.
GetRayIntersect	This function is the equivalent of our 'CCollision::RayIntersectPlane' function and is implemented in the same fashion. This function is called in many places, but the primary use in this lab project is during the splitting of polygon data. We will discuss where and when this splitting occurs later in this chapter.
SameFacing	During our coverage of both the node and solid leaf BSP compilation techniques in this chapter's textbook, we discussed how crucial it was that we pay special attention to the direction of the potential splitting polygons – in relation to the node planes – within the coplanar cases of our compilation techniques. This function simply accepts another plane as input and returns true if both planes point in the same direction, or false if they point in opposing directions. This should simplify several of the coplanar tests we would have to write, to a relatively simple call to this function.

We will be relying on the functionality of this class in many places throughout the development of the pre-processor application covered in this lesson. It is important that you understand how each of the member functions outlined in the above table work. If you are unsure of any of these procedures it would be a good idea to refer back to our previous coverage of these functions in addition to studying the new source code for the 'CPlane3' class found in the project files 'CPlane.h' and 'CPlane.cpp'.

- **CBounds3**

This class is a new concept and one which is not based on any structure or class that currently exists within the D3DX API. This axis aligned bounding box class wraps much of the

functionality we have come to rely on in our spatial partitioning compilation classes. In previous lab projects we have represented a bounding box using two separate 'D3DXVECTOR3' structures which describe the minimum and maximum world space extents of that bounding volume respectively. The 'CBounds3' class – the source for which can be found in the files 'CBounds.h' and 'CBounds.cpp – stores this same information in a single object. Because this bounding box information is now wrapped in a class of its own, there are several bounding box related operations that have been previously discussed which have been included in this class to provide a more integrated approach. The most important of these new member functions are described in the following table:

Member Function	**Description**
GetDimensions	This function calculates and returns the overall dimensions of the axis aligned bounding box volume on each axis. This is achieved simply by subtracting the minimum box extent values from those of the maximum box extents. As a result, this function returns a single 'CVector3' object in which the value of each component of that vector describes the dimension of the bounding box on each of its individual axes.
GetCenter	In order to perform many operations with a bounding box, we need to be able to find its center point. Because of the nature of an axis aligned bounding box, this can be determined using the following simple formula: '(Min + Max) / 2.0'. This function returns the vector that describes the position in space of the box centre point using the same process.
CalculateFromPolygon	In many cases in the past we have manually built our bounding box extent values by looping through the vertices in each individual polygon, growing the extent values if any vertex is positioned outside the box already described. Because this can be a lengthy and laborious coding exercise, this utility function wraps this process and allows us to specify a series of vertices with which it should build the bounding box values. In addition, this function also accepts a 'Reset' parameter that – when set to false – allows us to repeatedly call this function passing in a different polygon to each call and have it accumulate the results at each stage.
IntersectedByBounds	This function is our new math library's implementation of the previously discussed 'CCollision::AABBIntersectAABB' function. Put simply, this function accepts another 'CBounds3' class as an input parameter and returns 'true' if the space described by these two bounding boxes is found to overlap. Conversely, when they are not found to be intersecting, this function will return false.

PointInBounds	Our application has needed to determine whether a point falls within the space described by an axis-aligned bounding box on several occasions. In our previous lab projects this information was made available via a call to the 'CCollision::PointInAABB' procedure. This function accepts a single 'CVector3' input parameter containing the location that we would like to test against this bounding box. If the point is found to be within the bounding box extent vectors then this function will return 'true'. In all other cases, a resulting value of 'false' will be returned.
Reset	This function can be called in order to reset the bounding box member extent vectors to a set of values that can be used as the basis for the construction of a new bounding box. When initializing the axis aligned bounding box extent vectors, we will typically set each of the components of the maximum extent to a large but negative value and each of the components of the minimum extent to a large positive value. This ensures that any positional vertex or vector we consider during the calculation of the actual extent values will be taken into account. This function should be called if you plan on manually constructing the bounding box values.

This concludes the discussion of our new replacement math library. You should already be very familiar with almost all of the functionality exposed by these new classes but it was important that you understand the basic layout of each class in order to more easily follow and understand the compilation techniques we will be developing in this lesson. Before we continue on to our discussion of the code for lab project 16.2, there are one or two remaining differences that must be outlined in the tree building concepts we have been developing to date.

File Based Data Structures

One of the other key differences found when comparing this new compilation tool to the compilation techniques we have implemented to date are the means by which data structures are linked together. Each of the compiler classes we have implemented so far has been integrated directly with the game application itself. In these cases, we have been building the data structures intended for use directly by the game framework classes. In our new tool, we are building structures which are to be written out to file rather than simply maintaining that data in a memory oriented fashion.

In the past, we have often maintained links between data structures by making use of a direct pointer variable. As an example, the node classes provided by the various spatial hierarchy types we have implemented each store a pointer to other physical heap allocated child node and leaf structures. As we know, pointers are essentially variables which store the numeric starting address to the memory location in which another variable, class or structure has been allocated. When writing data to a file, we can not of course simply write the contents of a pointer variable as we would with a primitive data type such as an 'int' or 'char'. Instead we must interpret and write the data that is referenced by that pointer. While this seems simple enough, it can become quite difficult to untangle the referenced items and store them

in a manner which can be easily and efficiently reconstructed in the target application. This is especially the case when we are dealing with tree structures that are many levels deep.

When we are exporting data to file, we very often find data structures that have dependencies on one or more pieces of information – such is the case with most spatial hierarchies. In some cases there are certain pieces of information which are also shared between multiple objects. A classic example of this is with the spatial hierarchy node planes. In this case, we already know that in any individual branch of a BSP tree for instance, there should never be any other nodes that share the same plane. However there are very often many instances in which nodes are coplanar with another node *elsewhere* in the tree (e.g. further down the alternate branch of one of its parents). Due to this fact, it is entirely possible that we could select an individual node and find that it shares a plane with 10s or even 100s of other nodes in that tree.

Imagine a somewhat worse case scenario in which we found that our scene contained 10,000 unique nodes that were coplanar with exactly 9 other nodes in alternate branches of the tree. This is outside of those included in the original figure of 10,000. Even though a single plane structure occupies only 16 bytes for storage, these plane structures alone would consume 1,563kb within the file (10,000 * 10 * 16 bytes). Given that each of these structures defines exactly the same plane it should be clear that it would be much more efficient to simply store only one of these planes, and have each of our nodes simply reference that same plane data structure. If we were to do so, the planes in the above scenario would occupy only a fraction of the space at just 160kb (10,000 * 16 bytes). While this was admittedly an overly dramatic scenario, it should make the point clear that there are certain considerations that we may need to make when writing data such as this to file. Whilst it could certainly be argued that this same logic could be applied to our memory oriented data structures – and in this specific example scenario it would certainly be advisable – when working at runtime we have to balance efficiency and speed against the storage costs. However, speed of execution is rarely a concern when writing data to file and therefore, reducing storage space should certainly be one of our highest priorities.

Given the complexity of most tree structures in addition to the inherently interdependent data structure design and the potential need to reference a single piece of information from multiple items, it is often easiest for us to write data to file in contiguous blocks – whereby similar data structures are written together in one series. In essence this means that we would be exporting separate **arrays** of planes, nodes, leaves and other such data structures to file. In this case, we would simply have each item reference elements from within each of those ‘arrays’ using some form of indexing. As a byproduct of this we are also providing an easy and efficient way for our intended application to load the information back in, enabling it to load the data in a simple loop or even in some cases to read the whole block of data back into an array with a single read operation.

Although in the past we have previously constructed central arrays containing all of our leaf, polygon and detail area data structures, these existed only to provide a convenience to our spatial partitioning applications. We did this to allow our application to access all of the leaf and polygon data stored in the tree – via the base ISpatialTree interface – without having to perform a full scale traversal in order to access each of the individual pieces of information. In addition to this, our data structures referenced the components within the tree by utilizing a pointer based system. Using the node as an example – a data type which was not previously stored in a centralized array – child nodes were only ever referenced by their parent. The only means by which we could gain access to the data stored at any particular node

would be to start at the root, and traverse down the hierarchy using the child pointers stored at each subsequent node. As you might imagine, this would make it quite difficult to write the data to file for two main reasons. The first is that we have no centralized array to begin with and secondly, even if we did have such an array we would somehow have to convert any pointer which references a node within that array, into an index in order to write it to file. This would make our export procedure very complex with many procedures needed in order to search for locations within various different arrays before writing each component.

In our new file oriented tree compilation tool, the centralized arrays mentioned previously will serve as a much more integral concept. In this pre-processor application, the centralized arrays stored within the tree will become the *only* place that our data structures exist.

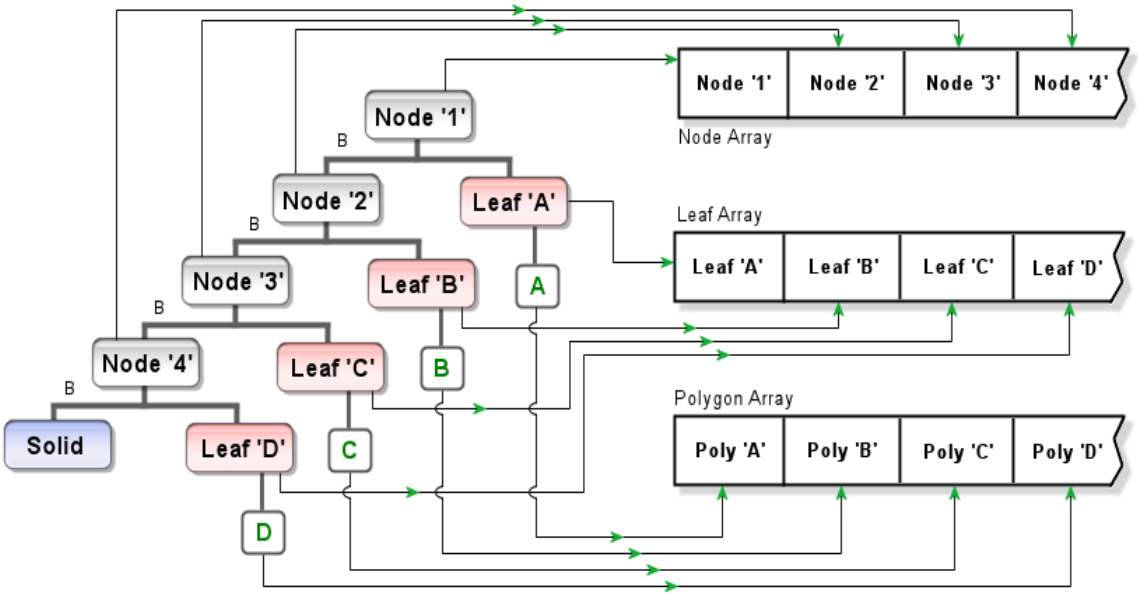

Figure 16.1

Figure 16.1 shows a number of arrays that might be used to store the tree information. We can see that each type of data structure used within the tree has its own centralized array. In addition, each instance of those data structures has a unique position within that array. Recall that we need to move away from the use of pointers in our data structures if we want to easily write the data to file. The upshot of reorganizing our compilation code in this way is that we can begin to use indices that direct us to the correct element in each of these arrays rather than using pointers (physical memory addresses):

```
class CNode
{
public:

    // Public Variables for this Class
    int Front;
    int Back;

    ...
};
```

This pseudo-class demonstrates how we might employ the use of indices within one of the most common tree data structure types. In previous implementations, the child member variables stored within the node were direct memory pointers. In this case, these pointers have been replaced with simple numeric variables of the type 'int'. If we ensure that any new data items are placed into a centralized array at the same time as we allocate that item it is guaranteed that we will have access to the correct array index which can be stored in the data structure itself. The following pseudo-code is an example of how this might be achieved.

```
// Allocate new node
pNewNode = new CNode;

// Setup new node data
...

// Attach to the front using its position as
// it will exist in the array (the next item)
pCurrentNode->Front = m_Nodes.size();

// Add the node to the centralized array
m_Nodes.push_back( pNewNode );
```

In this example we are creating a new front node in preparation to recurse into a new level of the tree construction process. In this code, the first thing we do is to allocate a new node. Imagine at this point that we had already stored 10 nodes in an STL vector called 'm_Nodes'. Recall that array indices are zero-relative which means that the first element in the array can be accessed using index '0', and likewise the 11th element by using index '10'. The very next line of code retrieves the *current* size of the node vector. Because we have not yet added the new node to the array – which as we know contains ten nodes already – the return value from the STL vector 'size' call will be 10. The final line in this example adds the new front node to the end of the vector as the 11th element in the array. Given the zero-relative nature of array indexing it should be clear that the index of '10' stored in the previous line is correct.

There are other types in addition to the node which we will of course have to adapt to this file friendly tree organization approach. These include both the leaves attached to the nodes and the polygons stored within those leaves. With the leaves in particular, we previously created unique child 'leaf-nodes' within our memory based tree structure whose primary purpose was to store a pointer to the leaf that existed there. During the compilation process, these leaf nodes were attached to the relevant child pointer of the current node enabling us to greatly simplify our traversal and construction logic. When written to file, these additional leaf nodes serve no informational purpose and as a result the BSP tree building class we will construct in this lesson does not include these additional nodes. Instead we will be taking advantage of the transition from pointers to indices in order to link the leaves directly to the parent node structures.

The easiest way to achieve this is to use a *signed* data type for the child index member variables stored within the node structure. In the previous example of how we might declare our index based node structure we used the signed type 'int' for the 'Front' and 'Back' member variables. As we know, a node needs to be able to reference two types of children. These are either another child node, or a leaf structure. A 32 bit *unsigned* integer variable is capable of storing values up to and including 4,294,967,295. Since it is not possible for us to ever create anywhere near this number of nodes, it is

safe to assume that we can use a *signed* data type which is capable of storing values between +/- 2,147,483,647. Using standard indexing procedure we know that we can use the positive index stored within a child member variable of a node to reference into our node array. However, since we are using a signed data type we can also use the negative range of values to signal to our traversal procedures – as well as our level import process – that once we convert it back to an absolute value, this member variable is an index into the *leaf* array instead.

One small caveat that you may have already spotted in this procedure is that when indexing into an array the first element is accessed with an index of 0. However, if the node's child index stores a zero value then there would be no way to know whether it is pointing to the first element in the leaf array or that same element in the node array. As a result of this we must make a slight adjustment to the signed / unsigned index ranges used such that nodes are indexed with values of 0 and above, and leaves are indexed using values from -1 and below. Take a look at the code below in which we show an example of how we might generate and store the index for a leaf in its parent node, using this signed / unsigned concept.

```
    // Allocate new leaf
    pNewLeaf = new CLeaf;

    // Setup new leaf data
    ...

    // Attach to the front using its position as
    // it will exist in the array (the next item)
    // but altered such that we use our negative range
    pCurrentNode->Front = -(m_Leaves.size() + 1);

    // Add the leaf to the centralized array
    m_Leaves.push_back( pNewLeaf );
```

As before, we first create our leaf object and set up its values as necessary. Again, on the next line we want to link this leaf to the current node's front by storing an index. In this case, imagine that we have not yet stored any leaves in the 'm_Leaves' vector. Because of the fact that the vector's 'size' function will return a value of 0 at this point, when we add 1 to this value and invert it's sign we would be storing a value of '-1' in the current node's front child member.

So, given that we are now storing indices in our node structure to both its child nodes in addition to directly referencing leaves, how might we go about turning these values back into physical array indices? The following pseudo-code example demonstrates just this.

```
    // Is there a leaf or a node in front of the current node?
    if ( pCurrentNode->Front < 0 )
    {
        // Get the leaf index
        int LeafIndex = abs( pCurrentNode->Front + 1 );

        // Retrieve the leaf
```

```
        CLeaf * pLeaf = m_Leaves[ LeafIndex ];

        ...

    } // End if leaf
    else
    {
        // Get the node index
        int NodeIndex = pCurrentNode->Front;

        // Retrieve the node
        CNode * pNode = m_Nodes[ NodeIndex ];

        ...

    } // End if node
```

As you can see, we can now detect whether a node is referencing a child leaf or another child node by testing the sign of the index stored in the relevant child member variable. If this value is found to be less than zero, then the variable is referencing an element into the leaf array. Conversely if it is found to be greater than or *equal to* zero, then we know that this must be referencing a child node. Due to the ranges that were chosen to describe each type of reference, when this child is found to be a node, we need do nothing with the index stored there. In this case, we can simply use the index to retrive the node from our centralized array directly. If it is a leaf however, we must undo the adjustment to its value that we made when storing the index, as well as flipping its sign such that we have an absolute index value. This is done in the following line of code.

```
int LeafIndex = abs( pCurrentNode->Front + 1 );
```

In this line we take the current value stored in the node's child member variable and add 1 to it before we retrieve the absolute of that resulting value. It may initially seem strange that we are again adding 1 to this value as we did when we were calculating the index for storage. Remember however, that the entire result of the original addition was negated just before we stored it. Before we move on, let us just test this code by using some real values. If you are not sure at this point exactly how this procedure works, then following through the example using real values should be helpful. The first example below demonstrates the initial calculation of the negative value that will be stored in the node when referencing a leaf. For the purposes of this example, assume that the leaf array already contains 10 other leaves.

```
Front = -(m_Leaves.size() + 1)

=
Front = -( 10 + 1 )

=
Front = -( 11 )
```

As you can hopefully see, even though our node will need to index into the leaf array using an index value of 10, we actually store a value of -11 in the node. Obviously this means that if we needed to index into the first element in the leaf array (index 0) that this same logic would generate a value of -1 as

we discussed previously. Let us finish off this example by retrieving the correct index based on the value that we have just calculated.

```
LeafIndex = abs( Front + 1 )

=
LeafIndex = abs( -11 + 1 )

=
LeafIndex = abs( -10 )

=
LeafIndex = 10
```

Following this example through demonstrates that the returned index value references into the leaf array using an index value of 10 as we had expected.

Although this does add a level of complexity to both our compilation and import processes, we are able to eliminate a number of nodes equal to that of the number of leaves from the resulting file. Given that the storage of an average node data structure might be in the area of 36 bytes per element or more, this would save us over 100kb in our final export file for a level containing 3000 leaves. Since these nodes do not give us any particular benefits within the file itself, this is an important storage optimization.

With these alterations made to the memory based structure we have previously used in our run-time projects, we should be able to quickly and efficiently export the tree data to file. Although the exact implementation details will ultimately be discussed as we move through the code and structures of our new application, this section should have made you aware of the implications of and reasons why we have moved to an array and index solution in our new pre-processing tool. As we move forward with this lesson, we will discover that our compiler tools make heavy use of this concept throughout and – as mentioned – will be heavily biased toward the way in which it will finally be exported. Before we finish up this discussion thefore, let us take a look at the final layout of the data as it will be written to file when saving the leaf tree information.

Table 1: Plane Data Layout

Name	Description	Type	Size (in bytes)
PlaneCount	The number of planes that have been written to file.	unsigned long	4
Repeated for each plane as described by the previously written 'PlaneCount'			
PlaneNormal	The normal of the plane being described.	CVector3	12
PlaneDistance	The plane distance / 'd' component of the plane.	float	4
End of 'PlaneCount' repeat.			

Table 2: Node Data Layout

Name	Description	Type	Size (in bytes)
NodeCount	The number of nodes that have been written to file.	unsigned long	4
Repeated for each node as described by the previously written 'NodeCount'			
PlaneIndex	Index into the previously written plane data / array that describes the plane used to construct this node. Note: Multiple nodes may reference the same plane.	long	4
BoundsMin	The minimum bounding box extents of this node. This describes the extent of all of the polygons contained below it in the hierarchy.	CVector3	12
BoundsMax	The maximum bounding box extents of this node. This describes the extent of all of the polygons contained below it in the hierarchy.	CVector3	12
Front	The front child index for this node. This can be either a positive or negative number as described previously.	long	4
Back	The back child index for this node. This can be either a positive or negative number as described previously.	long	4
End of 'NodeCount' repeat.			

Table 3: Leaf Data Layout

Name	Description	Type	Size (in bytes)
LeafCount	The number of leaves that have been written to file.	unsigned long	4
Repeated for each leaf as described by the previously written 'LeafCount'			
BoundsMin	The minimum bounding box extents of this leaf. This describes the extent of all of the polygons stored within this leaf alone.	CVector3	12
BoundsMax	The maximum bounding box extents of this leaf. This describes the extent of all of the polygons stored within this leaf alone.	CVector3	12
PVSIndex	This is used in the next lesson, for now it should be considered as a 'reserved' data area and should be written with a value of '0'.	unsigned long	4
PolygonCount	The number of polygons that are stored within / referenced by this leaf.	unsigned long	4
PortalCount	This is used in the next lesson, for now it should be considered as a 'reserved' data area and should be written with a value of '0'.	unsigned long	4
Reserved	This value is literally reserved for later use / expansion. This should be written with a value of '0'.	unsigned long	4
Repeated for each polygon as described by the previously written 'PolygonCount'			
PolygonIndex	Index into the main polygon array that was written into the **first** mesh in the IWF file.	unsigned long	4
End of 'PolygonCount' repeat.			
End of 'LeafCount' repeat.			

Navigating the Compiler Project Source Code

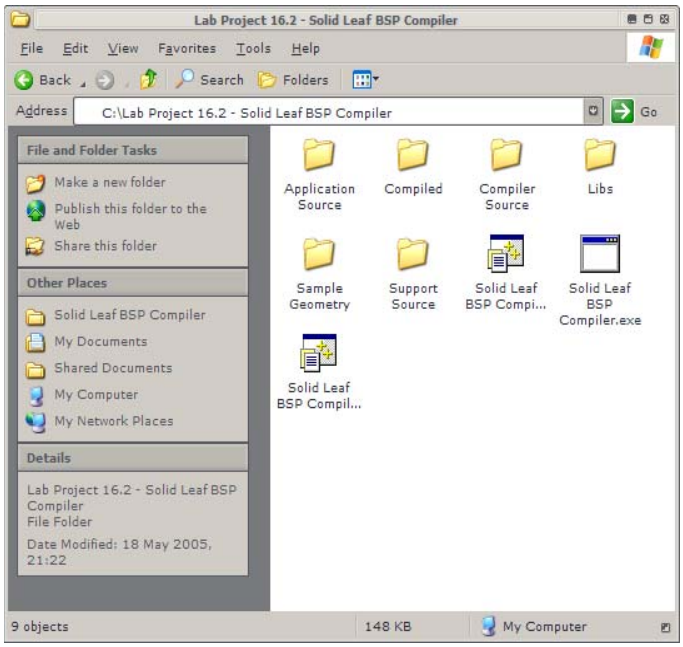

Figure 16.2

Because we are building a brand new application in lab project 16.2 and one that is not based on the previous demo framework we have used in the past, there are some changes to the layout of the project directory structure. As was mentioned previously, the pre-processing application that we are building in this lesson is designed to be as modular as possible. This will allow us to add additional features and compilation procedures to the application in the future without having to rewrite large portions of code. In addition to this, we have tried to design a portable system whereby the core compilation portions of the application is completely separated from whatever front end interface might be desired. In certain cases there are situations in which some form of feedback is required by the application. One example of this is how the application displays the current progress and outcome of each compilation task. As we have discussed, some compiler computations can take a long period of time to complete. If our application simply displayed a blank screen until the scene was completely processed, the user might reasonably believe that the application had stopped responding. Due to the fact that we want to keep the compiler completely separated from the application front end, and that only the compiler is really in a position to inform the user about how each task is progressing, this presents us with a problem. We can overcome this situation however, by defining a base interface – which we have named 'ILogger' – that both the front end and the compiler are aware of. The application is then responsible for deriving a class from this interface and its defined functions, which displays the logging information in any way it requires. Because the compiler uses the abstract 'ILogger' interface to output any progress information, we preserve the independence of the compiler and our ability to insert it into any application.

At this point, rather than become distracted by a comprehensive and potentially complex full blown GUI front end, we have opted for a relatively simple console based framework in which we will insert the compiler code. We have also developed a derived logging class which simply outputs the progress information directly to the console window itself.

It is clear that we have two separate components at this stage. We have the application framework and its associated front end, in addition to the compiler logic and compilation process source code. We have also introduced several new support classes however – such as our new vector, matrix, AABB and plane classes – that bring several new header and source files with them. In an effort to ensure that our pre-processing tool source code can be more easily managed, each of the source components that make up our application has been separated into various directories.

Although it should be relatively easy to find your way around the source code folder, the following list outlines the contents of each of the directories in the new project structure in addition to the intention behind them should you need to add additional source files of your own.

Application Source

This directory contains the main source files and headers related only to the application framework and interface front end that will create and call the compiler classes and functionality. The intention behind this directory is that it should contain the source files and header files that define the application entry point, any sort of display procedures, graphical user interface classes and anything which is not part of the core compiler itself. In this lab project, there are two main source file and header combinations located in this directory:

Main.cpp & Main.h
These source files define the application entry point (the *WinMain* function) which is ultimately responsible for setting up and calling the compiler class functionality. These files are relatively simplistic but are responsible for handling how the compiler is set up as well as the main logic of the application execution.

LogOutput.cpp & LogOutput.h
These source files define the derived 'ILogger' class named 'CLogOutput'. This class's purpose is to output the progress information to the application's console window, as reported by each compilation process. This class implements the pure virtual functions declared by the 'ILogger' interface.

Compiler Source

This directory is the home of our compiler application's core processing source code. Any source code files relating to the individual compilation tasks such as the BSP compilation, hidden surface removal and T-Junction repair procedures included in this lab project can be found here. This is also the directory into which we will place any additional modules that will be developed in later lessons. In addition, all source files relating to the import and export of the scene and compiled data can be found here.

Compiler.cpp & Compiler.h
These files define the class 'CCompiler'. This class exposes the primary compiler API that our application front end will use in order to perform the core tasks of our pre-processing tool. For instance, this class includes functions which allows the application choose which compiler tasks to perform, to set any applicable compiler options and to instruct the compiler component to load and save a particular file. The application will never typically work directly with the individual compiler process classes (outlined shortly), but will instead work through the interface defined by this class.

CompilerTypes.cpp & CompilerTypes.h
There are many classes defined by these files which include items such as 'CMesh', 'CPolygon' and 'CVertex'. In each case, the classes and structures defined here are dependencies of the compiler code. In most cases, these classes are used to house any data that has been loaded from the scene file specified

by the application. Also defined here are the various options structures and miscellaneous definitions that are used by the application to setup the various compiler processes.

CFileIWF.cpp & CFileIWF.h
The 'CFileIWF' class is defined in these files. This class is responsible for loading and saving our scene data both before and after compilation. Although we have previously been using the 'CFileIWF' class as provided by the static 'libIWF' import library, in this project we also need to include the file export logic that is executed once the compilation procedures have completed and the application requests that the compiled file is saved. In addition we are using several custom classes – such as the custom mesh, polygon and vertex classes discussed later. Rather than complicate matters by using intermediate data structures for importing and exporting the tree data, we will implement the 'CFileIWF' class directly here in order to extend it and allow us to use our custom classes directly.

CBSPTree.cpp & CBSPTree.h
The classes defined in these files are the very foundation of everything we will be developing in both this lesson and the next. These classes include our main solid BSP leaf tree compilation class (CBSPTree) in addition to the leaf (CBSPLeaf), node (CBSPNode) and other associated classes. The 'CBSPTree' class also serves as the central hub for storing most of the information which we will be building throughout the various compilation tasks.

ProcessHSR.cpp & ProcessHSR.h
Because our compiler tool is designed to be modular, each of the individual compilation processes is separated out into individual source files and classes. These files define the 'CProcessHSR' class which is home to the hidden surface removal processing logic. This module is created and called by the aforementioned 'CCompiler' class during the overall build process.

ProcessTJR.cpp & ProcessTJR.h
As with the HSR process, the T-Junction repair step is segregated into its own modular class. This class as defined within these files is named 'CProcessTJR'. Again, this class is instantiated and executed by the 'CCompiler' class during scene compilation. Although we will implement additional processing modules in the future, this process is generally the one that will be executed as the very final step before saving.

Support Source

The support source directory contains all of the miscellaneous supporting classes used by the compiler in addition to any header files which may be common to both the application front end and the compiler classes. Recall that we previously discussed the inclusion of a new supporting math library which included the classes 'CVector3', 'CVector4', 'CMatrix4', 'CPlane3' and 'CBounds3'. Although not listed below, this directory also contains the source code implementations for each of these new classes.

Common.h
This is the common header file which links the front end application with our compiler core. This file includes several macros and other useful definitions that are used by both components of our application. It also includes certain other headers which will need to be in place for the compiler to

integrate into the application without issue. This header is also one in which the 'ILogger' interface is declared, from which the application should derive its progress display class.

AppError.h
Last but not least is the header that contains all of the potential error definitions used by the compiler and available for use by the application. Modeled after D3D's HRESULT error codes, these types of error descriptions should be familiar to you. Although we are using a 'result code' system similar to that of D3D in this application, for the most part these errors are not used as return codes. Instead the compiler makes extensive use of exception handling. As a result, whenever we need to handle errors which occur within the compile process, we include a catch block using an argument type of 'HRESULT' which is also defined here.

Now that we are familiar with the new project layout and design, it should be a little easier to navigate to the various portions of the source code with greater ease as we progress onto the implementation of our new tool. As such, let us finally move on to the next section in which we will begin to focus on the actual implementation of some of the classes we have discussed.

Scene Data Classes – CompilerTypes.cpp

Before we can realistically move on to the implementation of our new compiler application, there are several scene related classes that need to be discussed first. Throughout the development of the many lab projects we have built in the previous lessons of this series, we have created many classes whose purpose is to create, store and render the scene data. These included classes such as CMesh, CPolygon and CVertex. In each of these cases, a lot of thought had to go into developing a system which could be easily integrated into our application and was able to construct data in a hardware friendly fashion. We also developed several other structures and classes that stored and processed any secondary data such as texturing and material information. Obviously, these previous applications were heavily biased toward the rendering of a scene. As a result, a lot of work had to be done in order to prepare the information contained within the 'IWF' scene file that would ultimately allow our application to display the game world correctly.

In this application the primary focus is on importing and storing the data for use by the various compilation procedures. As a result we are able to simplify most of the scene data concepts we have previously employed, into one that simply provides a logical place for storage and that grants our tool easy access to that information. There is an additional step in this new project that will make use of the scene data however. In this tool we must also *save* the scene to file. This is something that we have not had to consider in those previous rendering applications. Because of this additional process, there are several pieces of information that must be imported and maintained that have no other purpose than to be exported back out to the resulting file. Some examples of this type of data are: surface textures and materials, any shader references that may be contained within the file and also scene entity information which we have come to rely so heavily upon in the various rendering projects.

The tools that we have traditionally used to load all of our scene information have always been those exposed by our static IWF import library. Recall that the main functionality needed for the importation of the scene data from the IWF file was made available to us through the 'CFileIWF' class and its support structures. We have previously touched upon the fact that we will be implementing our own

custom version of that class in this project. We need to do this because there are several custom scene data storage classes which are being utilized within the tool. In addition, our application is of course now required to add export functionality to this file handling class. Shortly we will be discussing the classes and structures we will implement for the data types used directly by our new compiler application. However, rather than re-implement each of the data classes that our application will **not** make direct use of – e.g. materials, entities, textures and shader information – we will rely once again on the data classes exposed by the import library. Since we have previously covered most of the support classes provided by the libIWF library, let us just quickly recap on each of the data classes that our pre-processor tool will employ.

- **iwfMaterial**

```
class iwfMaterial
{
public:
    // Constructors & Destructors for This Class.
     iwfMaterial( );
    ~iwfMaterial( );

    // Public Variables for This Class
    char           *Name;           // Material Name (if applicable)
    COLOUR_VALUE    Diffuse;        // Diffuse reflection component
    COLOUR_VALUE    Ambient;        // Ambient reflection component
    COLOUR_VALUE    Emissive;       // Emissive reflection component
    COLOUR_VALUE    Specular;       // Specular reflection component
    float           Power;          // Specular reflection power ratio
};
```

This is a class that we have already encountered in previous demo lab projects. The data members declared within this class closely represent those defined by the Direct3D 'D3DXMATERIAL9' structure and is used almost exclusively in the calculation of scene lighting information. The member variables declared by this class include the ambient, diffuse, specular and emissive surface reflection properties with which you should be familiar. Each of these material properties are common throughout many of the available vertex and pixel lighting techniques and should therefore also be portable to other immediate mode rendering APIs. As with each of these data storage classes, the declaration for this class can be found in the 'iwfObjects.h' header file in the project's 'Libs' subdirectory.

- **iwfEntity**

```
class iwfEntity : public IIWFObject
{
public:
    // Constructors & Destructors for This Class.
             iwfEntity( ULONG Size );
             iwfEntity( );
    virtual ~iwfEntity( );

    // Public Variables for This Class
    UCHAR           AuthorIDLength; // Length of the author ID code
    UCHAR           AuthorID[255];  // Actual author ID data.
```

```
    USHORT          EntityTypeID;   // The Type Identifier for the entity.
    char           *Name;           // Entity Name
    MATRIX4         ObjectMatrix;   // The matrix assigned to the entity.
    ULONG           DataSize;       // The size of the entities data area
    UCHAR          *DataArea;       // The entities data area.

    // Public Member Functions Omitted
};
```

As with the aforementioned 'iwfMaterial', we have seen and used the 'iwfEntity' class many times in previous lab projects. As we know, the entity type is a generic object concept that can be used to store many different types of information. We have used this class for objects such as lights, fog, trees, references, external meshes / characters, terrains and even a skybox. In this application we will not interact directly with the imported entity objects. This class will simply be used to store the imported entity data until such time as the scene has been processed, and the application chooses to export the scene to another file.

- **iwfTexture**

```
class iwfTexture
{
public:
    // Constructors & Destructors for This Class.
             iwfTexture( ULONG Size );
             iwfTexture( );
    virtual ~iwfTexture( );

    // Public Variables for This Class
    UCHAR   TextureSource;  // Information about where the texture is
    char   *Name;           // The texture name.
    UCHAR   TextureFormat;  // The format of the internal texture if used
    USHORT  TextureSize;    // Size of the TextureData array
    UCHAR  *TextureData;    // The data being referenced

    // Public Member Functions Omitted
};
```

Although we have not used the 'iwfTexture' class itself in the past, this class is just a more object oriented version of the 'TEXTURE_REF' structure that we have used in almost every lab project we have developed from chapter five onwards. This class is designed to store various pieces of texturing information, but the most common use is one in which each instance stores the filename of a scene texture to be applied to the appropriate pieces of level geometry that reference it. As with the material and entity information, this application will not interact directly with objects of this type. This data will simply be imported along with the scene data, and exported again when the processed scene is saved.

- **iwfShader**

```
class iwfShader
{
public:
```

```
    // Constructors & Destructors for This Class.
             iwfShader( ) {};
    virtual ~iwfShader( ) {};

    // Public Variables for This Class
    ULONG           Components;     // Which components are included
    iwfScriptRef    VertexShader;   // The vertex shader if included
    iwfScriptRef    PixelShader;    // The pixel shader if included
};
```

We have not previously used the shader classes or structures exposed by the libIWF import library, but they are maintained within this application in order to allow for future enhancement. The 'iwfShader' class itself does not store any specific pieces of shader information. Instead it declares two members named 'VertexShader' and 'PixelShader'. Each of these members is of the type 'iwfScriptRef' outlined below. Once again this information is only imported for the purposes of being transferred to the export file once the scene has been compiled.

```
class iwfScriptRef
{
public:
    // Constructors & Destructors for This Class.
             iwfScriptRef( ULONG Size );
             iwfScriptRef( );
    virtual ~iwfScriptRef( );

    // Public Variables for This Class
    UCHAR   ScriptSource;   // Information about where the script is
    char   *Name;           // The script name
    USHORT  ScriptSize;     // Size of the ScriptData array
    UCHAR  *ScriptData;     // The data being referenced

    // Public Member Functions Omitted
};
```

This class is intended to reference script files or script data in much the same way as the 'iwfTexture' class references texture files. While this structure is used independently in other locations within the IWF specification, the script reference concept is used only for storing shader file references in this lab project.

If you would like further information about the various structures, flags and concepts surrounding each of these IWF data classes it would be a good idea to refer to the IWF SDK documentation available from the classroom download area.

The CVertex Class

We should already be extremely comfortable with the concept of a vertex class because we have used them extensively since the very first chapter. In each of the lab projects to date however, our vertex class has been used primarily for storing data in a format best suited for the purposes of rendering, or building vertex buffer data. In this application we are focused more on the storage and ease of use aspects due to the fact that our pre-processing tool will have no rendering component. As a result, there are a few

changes which have been made to the vertex class concept that should make the task of developing this new application a little easier.

```
class CVertex : public CVector3
{
public:
    // Public Variables For This Class
    // (X/Y/Z components inherited)
    CVector3    Normal;    // Vertex Normal
    float       tu;        // Vertex U texture coordinate
    float       tv;        // Vertex V texture coordinate
```

The first thing to notice about the declaration of our new 'CVertex' class is that it is derived from one of the new math support classes discussed earlier in this chapter. As you can see, we derived this class from 'CVector3' which if you recall contains most of the vector functionality previously offered by the D3DX math component. By deriving the vertex class from 'CVector3' in this way, our compiler component is able to interact with either a vertex or vector object in exactly the same way. In addition, we are able to pass any *vertex* object into a function that is expecting a vector typed parameter. As an example, using this class hierarchy we might perform a dot or cross product on the vertex data directly. We could also classify a vertex against a plane without any of the confusing casting operations that we would previously have included with our standalone vertex structure. In essence, our vertex class is simply extending the functionality of the vector, allowing us to integrate the two concepts more easily.

In this portion of the class declaration it is important to notice that the X, Y and Z components of the vertex have not been included as data members. This is due to these positional components having already been declared by the new three dimensional vector class from which it is derived. By doing so, the vertex class inherits these three public member variables from the vector and as a result they should not be declared again. This is the very thing that allows our vector class functionality to work directly with the positional component of our vertices.

Although there is no direct implementation of the 'CVertex' within any source module (.cpp file) – outside of those member functions inherited from the vector class – there are several constructors defined directly within the 'CompilerTypes.h' header file:

```
    // Constructors & Destructors
    CVertex( ) { x = y = z = tu = tv = 0.0f; }
```

The first of these is the default constructor for this class. This function accepts no input parameters and simply resets each of the vertex component values to zero. Notice how in this function we are setting the inherited vector component values to zero in addition to those declared by the vertex class itself (the texture coordinate values tu & tv). One thing that may appear to be lacking in this constructor function is the setting of any default values for the 'Normal' member. The vertex normal is declared using the 'CVector3' class however, which defaults the X, Y and Z components to zero automatically within its own default constructor. The normal component's constructor will be called implicitly when an instance of the 'CVertex' class is created at any point within the application and as a result we need node duplicate this functionality here.

Of course, the condition under which the default constructor is called is only one of the many situations in which we may want to construct a vertex object. In many of our previous lab projects we have included several vertex class constructors which allow us to initialize its values quickly and efficiently. Similar alternate constructors have also been included in this application for the same reason.

```
    CVertex( float _x, float _y, float _z )
    {
        x = _x; y = _y; z = _z; tu = tv = 0.0;

    } // End Constructor
```

This constructor should be relatively self explanatory. It accepts three parameters which are simply used to initialize the positional portion of the vertex component. We can use this constructor in several situations including either object initialization or assignment as shown below:

```
// Stack Allocation / Initialization
CVertex Vertex1( 10.0f, 5.0f, 0.0f );

// Heap Allocation / Initialization
CVertex * pVertex = new CVertex( 10.0f, 5.0f, 0.0f ) ;

// Assignment
CVertex Vertex2 = CVertex( 10.0f, 5.0f, 0.0f ) ;
```

The other vertex components, such as the vertex normal and texture coordinate values, are set to an initial value of zero in this case.

The next constructor provided by this class is one that accepts a parameter of the type ‘CVector3’ which is again used to construct only the positional component of the vertex.

```
    CVertex( const CVector3& vec )
    {
        x = vec.x; y = vec.y; z = vec.z; tu = tv = 0.0f;

    } // End Constructor
```

This constructor is one of the most important for providing a level of interoperability between the new vector and vertex classes. As with the previous constructor we can use this for both initialization and assignment of the vertex data as shown below:

```
// Vector
CVector3 SomeVector( 10.0f, 5.0f, 0.0f );

// Stack Allocation / Initialization
CVertex Vertex1( SomeVector );

// Heap Allocation / Initialization
CVertex * pVertex = new CVertex( SomeVector );

// Assignment
CVertex Vertex2 = CVector3( 10.0f, 5.0f, 0.0f );
```

```
CVertex Vertex3 = SomeVector;
CVertex Result  = SomeVector.Cross( CVector3( 0.0f, 1.0f, 0.0f ) );
```

The latter of these examples – in which we assign the value of a vector directly to the vertex – is probably the most common situation in which this constructor will be utilized. The bottom-most assignment example shows how we might return the result from a vector cross product operation and store it directly within a vertex object. In this case, the vector assignment constructor will be called and the positional components updated. The remaining vertex components are initialized to a default of zero as before.

There are obviously several other components within our vertex class that may need to be initialized along with the vertex position. In response to this our vertex class also includes constructors which accept additional parameters for this purpose.

```
    CVertex( float _x, float _y, float _z, float _tu, float _tv )
    {
        x = _x; y = _y; z = _z; tu = _tu; tv = _tv;

    } // End Constructor

    CVertex( float _x, float _y, float _z, const CVector3& _Normal,
             float _tu, float _tv )
    {
        x = _x; y = _y; z = _z; Normal = _Normal; tu = _tu; tv = _tv;

    } // End Constructor
};
```

The final two constructors in our vertex class provide the ability to initialize the remaining vertex data not covered by those previously discussed. In each case there are additional parameters which match up to the data members exposed by the vertex. In the former of these final constructors, two new parameters labelled '_tu' and '_tv' are included. These will be used in circumstances in which the application would like to initialize the vertex using both positional and texture coordinate information. The latter of these two extends the first by including a parameter labelled '_Normal' which is used for the initialization of the vertex normal component.

There are of course many additional combinations of vertex class constructor which we may want to include at some point in the future. However, the constructors included here should be more than enough to provide our application with a flexible and comprehensive system by which vertices can be created, initialized and manipulated in many situations.

The CPolygon Class

With our vertex class fully defined, we now need a container in which they can be stored. As has been the case in many of the lab projects developed in previous lessons, the storage and logical grouping of the scene vertex information is the responsibility of the 'CPolygon' class. Although in some cases we have stored vertices directly within a mesh object – such as a 'CTriMesh' – this was almost exclusively built as a final step prior to its use within the rendering portion of the application.

As has been discussed in previous chapters, the spatial hierarchy compilation process that has been developed to date requires that the polygons used during the spatial partitioning phase are each convex winding ‘n-gons’. This is due to the fact that we often needed to split the polygon data, a process which is most easily achieved when the polygon data fits these criteria. In the case of the *polygon aligned* BSP tree however, the polygon concept plays a much more crucial role. Because the scene data itself will be used as the basis of the separating planes and nodes within the spatial hierarchy, it is vital that we ensure that each of the individual polygons within the scene remain separate. If we were to store all of the vertices directly within a mesh type container, indexing them using a triangle list for instance, this would obviously make the construction of our polygon aligned BSP tree much more complex.

Unlike these previous applications, we are now attempting to construct a much more comprehensive tool. In the past, the polygon class was often merely a means to an end in which vertex data was stored until it was used in the construction of hardware friendly vertex and index buffers for the purposes of being rendered. Our pre-processing tool on the other hand will need to work much more closely with various types of polygon data. As we add more features to the tool, there will be an increasing number of situations in which a particular compilation process will need to have access to additional polygon member variables, or even require an entirely custom polygon data structure. To see why this is the case we need only think back to the discussion of collecting polygons at the leaves in a polygon aligned BSP leaf tree.

At each stage in this compilation process, a polygon is chosen to become the separating plane for the current node in the hierarchy structure. Whenever a polygon is selected, it is removed from further consideration but it should still be passed down the relevant side in order to be collected in a leaf structure at a later stage in the process. This can be achieved very simply by exposing a custom ‘UsedAsSplitter’ flag from within the polygon itself for instance which is set whenever a polygon is chosen to become a node plane. As the already selected polygon is passed through the many levels of the recursive procedure, this flag can then be tested very quickly in order to determine if it should be used as a separating plane or not.

This is a prime example of a situation in which an individual compilation process might need an additional data member within the polygon structure. As you might imagine, there could be many circumstances under which this situation might arise as we add additional compiler modules to this tool. In the next chapter we will be introducing an additional compiler step that creates polygon-like constructs called portals. These portals are generated such that they describe the gaps in the geometry that exist between the leaf areas in the compiled tree. If you imagine that we had two rooms connected by a short length of corridor, the portal can be thought of as describing the holes that exist in each ‘doorway’ that connects each of the rooms to either side of that corridor. Essentially this will provide us with a form of leaf connectivity information that describes how each area of the scene connects to its one or more neighboring areas. In the case of the portal, we are only really interested in the shape of these portal polygons as defined by the vertices stored within them. In addition, this portal will need to store information about which leaf exists on either side of that portal polygon.

In the former of these example situations, we are required to add an additional member to the polygon class in order to store the ‘UsedAsSplitter’ information. With the later implementation of our portal compiler, there are yet more data members that are required and even some common concepts that

should not be included – such as texturing and material information – to keep memory requirements as low as possible. So, with the many types of polygon data requirements imposed upon us by this modular compiler concept, what is the best way to define our polygon class?

There are several ways in which this can be achieved. The first, and by far the simplest, is to add everything that we might need to our 'CPolygon' class, for every process that we implement, and simply not worry about any of the memory implications at all. While this certainly works in the majority of situations, it does mean that we will probably have to modify the polygon class every time we plug a new module into the compilation process. If our application makes copies of polygon members for any reason (such as during the splitting of that polygon into two new fragments), we will also have to ensure that the new members get duplicated across the board. As you might imagine, this could become a significant task and is somewhat in opposition to our wish for a modular pre-processing tool. Another way by which this can be achieved is for us to create a new polygon class for each process in which we need a custom data structure. This however, would require that much of the functionality which makes use of the polygon data and that which can be exposed by the polygon class itself – such as splitting, classification etc – may have to be duplicated for each type of polygon class that is implemented.

In this application we will be using a combination of both of these solutions in addition to borrowing one of the ideas we recently introduced in the new vertex class – that of utilizing class inheritance to make the process of development and future upgrades a little easier.

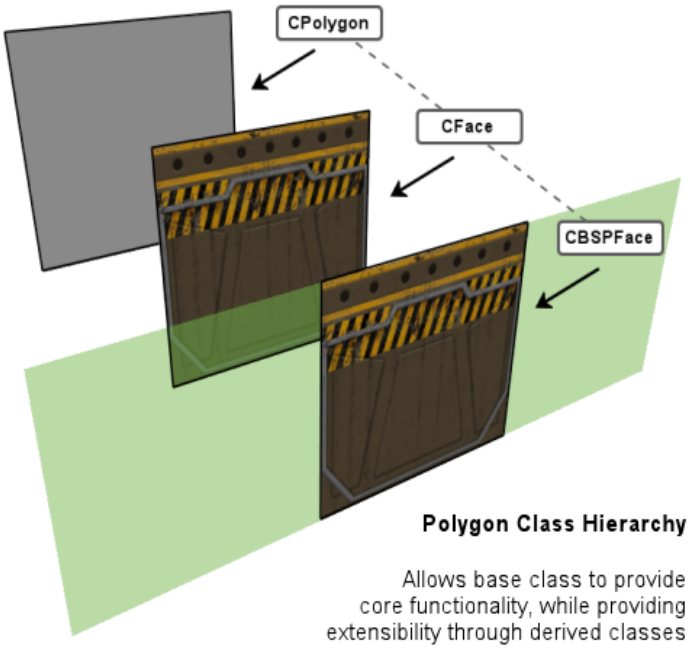

Figure 16.3

Figure 16.3 demonstrates how this type of hierarchical class design might work. In the top left of the image we can see our 'CPolygon' class which represents a plain polygon structure. In this class we would store only the vertex data required to define the polygon. This class would also implement several pieces of key polygon functionality such as the vertex storage and access functions, as well as the 'Split' function required by many of the spatial partitioning compilation techniques that we have already encountered. This base 'CPolygon' class would not declare any additional member variables or functionality that deal with additional surface properties such as texture or material information. By doing this we will later allow new structures to be derived from 'CPolygon' – such as the portal construct described earlier – that does not require any of these additional members, but does require the base functionality. With each of the vertex data members and management functionality provided by this base class, we can then derive a new class from 'CPolygon' in order to extend it for use in situations where additional rendering properties are required. This extended class is depicted in figure 16.3 by the polygon labelled 'CFace'. This class will include several additional

properties that we will need to store. Such properties might include texture, material and shader properties, surface normal, blending modes and any other additional data that we may want to use for rendering purposes in our runtime application. The final class in this diagram is depicted by the polygon labelled 'CBSPFace'. As you may have already guessed, this class is designed for use by our new BSP compiler during the spatial partitioning process. Because the BSP compiler will need to work directly with the *renderable* data, it makes sense that the new 'CBSPFace' class derives directly from the middle class 'CFace'. In this way, the custom BSP compiler polygon class will inherit any and all data members provided by its parent. It also allows us to extend this polygon class with any additional properties – such as our 'UsedAsSplitter' flag – that may be required **only** during BSP compilation.

We will discuss each of these classes in detail a little later on in this lesson. It is important however to understand the general design of our polygon data classes before we move on. Hopefully it is clear that by using a class hierarchy in this way, we are able to extend the functionality of the polygon class without the base class having to be aware of any of the additional properties required by any of the future processor modules. Each of these modules can simply derive a new class from the relevant base implementation which will automatically inherit the functions and data members defined by its parent thus sparing us from having to duplicate large amounts of functionality in each process specific polygon class.

With these general design concepts understood, let us move on to discussing the base 'CPolygon' class that is used within our new pre-processor tool.

```
class CPolygon
{
public:
    // Constructors & Destructors
             CPolygon( );
    virtual ~CPolygon( );

    // Public Variables for This Class
    CVertex        *Vertices;           // Polygon vertices
    unsigned long   VertexCount;        // Vertices in this poly

    // Public Virtual Functions for This Class
    virtual HRESULT Split( const CPlane3& Plane, CPolygon * FrontSplit,
                           CPolygon * BackSplit, bool bReturnNoSplit = false );

    // ** Non-Virtual Public Member Functions Omitted **

};
```

> **Note:** As with most class declarations shown throughout this workbook, member functions have been omitted where appropriate. Each of these omitted member functions is discussed in detail independently within that section.

As you can see, the class declaration for our base 'CPolygon' is relatively simple compared to those we have encountered in the past. We previously discussed how our base 'CPolygon' class would only contain the basic member variables and functionality required to define the polygon layout itself. This is in order to allow any individual compiler module to derive from the most appropriate base class. As a result, this declaration includes only two member variables.

CVertex *Vertices

The first and most important member variable declared within this class is a pointer that will store the address to our heap allocated linear vertex array. Each element in this array will contain a 'CVertex' object in the same way as we have seen implemented many times before throughout this course. Each of the vertices in this array defines the shape and boundaries of the convex winding 'n-gon' that is a pre-requisite for the compiler procedures that we will go on to implement shortly. Remember that our 'CVertex' class is derived from 'CVector3' and as a result we will be able to perform 3D vector operations directly on any element within this array.

unsigned long VertexCount

This member variable contains the value which describes the total number of vertices stored within the above vertex array. Although we have become accustomed to using STL vectors for the storage of data within recent lab projects – which would remove the need for a separate count variable – the vertex array stored here is most often static in size. As a result it is sometimes more efficient for us to handle the array allocation / resizing manually in this way.

The final important point to notice about the base 'CPolygon' class is how the 'Split' function has been declared. Although this function and its parameters should be very familiar to you at this stage, unlike the previous implementations this function has been declared as *virtual*. This allows us to safely include a new split function in any future polygon class in order to extend it with additional functionality. In addition, the class destructor is also declared as virtual to ensure that when the application deletes an instance of a polygon using a pointer of a *base* class type, the data members declared by the whole class hierarchy are correctly released. We will see how these concepts can benefit us when we move on to discuss the derived 'CFace' class in the next section.

CPolygon::CPolygon()

In this application, the default 'CPolygon' constructor does not have a great deal of work to do. As demonstrated in the following code, this constructor simply initializes the two base class member variables to ensure that they can be tested and used safely in later functions.

```
CPolygon::CPolygon()
{
    // Initialise anything we need
    Vertices    = NULL;
    VertexCount = 0;
}
```

CPolygon::AddVertices

This function allows the application to reserve space within the polygon for the required number of vertices. It can be called multiple times in order to grow the vertex array in each subsequent call. In each case, this function will return the index to the first vertex added to the array in that specific call. Although we have encountered this type of function several times in past lessons, we are using a different vertex concept in addition to a slight variation of the error handling concepts we are familiar with.

```
long CPolygon::AddVertices( unsigned long nVertexCount )
{
    CVertex *VertexBuffer = NULL;

    // Validate Requirements
    if ( nVertexCount == 0 ) return -1;
```

The only parameter accepted by this function allows the application to specify how many additional vertices we would like to add to the vertex array stored here. If the vertex array has already been created by a previous call, this will be the number of elements by which that array will be grown. As in the many functions we have implemented before, this parameter is validated to ensure that the allocation will not fail due to the application specifying a count of zero.

The first real task that this function must undertake is to allocate a temporary array of the correct size that will be used as the new vertex array.

```
    // Allocate brand new buffer
    try
    {
        VertexBuffer = new CVertex[ VertexCount + nVertexCount ];
        if (!VertexBuffer) throw std::bad_alloc(); // VC++ Compat
```

Here we allocate the memory that will be used to build a new vertex array of the requested size. We specify a number of vertices to be allocated which is made up of the number of additional vertices requested by the application (nVertexCount) added to the number of vertices that were previously stored within the polygon. This array of vertices will eventually replace the one currently stored within the polygon. Remember that because the 'CVertex' class contains a default constructor, there is no need for us to initialize the elements within the vertex array as we may have done in the past.

In previous implementations of the memory allocation functions, we have often simply tested the resulting value stored within the temporary buffer variable to see if it contains a 'NULL' pointer. In some version of Microsoft Visual C++ this was the means by which the 'new' operator signalled that an allocation failure had occurred. In later versions of Visual C++ however this behaviour was changed such that an exception will be thrown in this case. In order for us to support both methods of error detection we have wrapped the allocation of the new vertex array in a 'try' block. This ensures that if the 'new' operator throws an exception, we are able to handle this failure gracefully. To enable us to provide backwards compatibility however, we have included the aforementioned 'NULL' pointer test here also. In this case, we generate the same exception as we would expect the 'new' operator to throw. In doing so, we are able to support both types of error handling simply by including a single catch block.

```
        // If any old data
        if (Vertices)
        {
            // Copy over old data
            memcpy( VertexBuffer, Vertices, VertexCount * sizeof(CVertex));

            // Release the memory allocated for the original vertex array
            delete []Vertices;
```

```
        } // End if old vertices created

    } // End try block
```

This next portion of the code tests to determine whether or not vertex data already exists within this polygon. If this is found to be the case, then that vertex data is copied into the new vertex array to ensure that any vertices already inserted by the application persist. Notice how we copy only the amount of data that is contained within the *original* vertex array. This is important because we need to ensure that we do not attempt to read any memory past the end of the previously allocated block. This means that if our polygon previously contained 5 vertices, and our application was requesting space for an additional 5 then only those original vertices would be copied into the first 5 elements of the new array, leaving the last 5 elements in their default state.

The final task in this block of code is to release the memory (if any) that was allocated in a previous call to this function. If a vertex array already exists, we must release it to ensure that our application does not leak any memory when the 'Vertices' member pointer is overwritten with that of our new array.

With all of our memory allocation and initialization completed, there are no further exceptions which are likely to be generated within the context of this function.

```
    // Was an exception thrown?
    catch (...)
    {
        if (VertexBuffer) delete []VertexBuffer;
        return -1;

    } // End catch block
```

The catch block we are using in this function specifies an ellipsis in place of a specific exception handling type. What this basically means is that this catch block will capture any and all exceptions generated in the try block regardless of the type of exception thrown. Because the application is not interested in the specifics of any exception that may have occurred in this function, we simply release the memory allocated for the new vertex array – if the exception was thrown after the allocation line – and return a value of -1. Because the application is expecting a valid index in the range of 0 and above, this *invalid* index return code can be used to inform the calling function that the allocation was **not** successful.

At this point we have allocated a new vertex array that has enough elements to contain all of the original data that was stored in the array prior to this call, as well as those additional elements requested by the calling function. With the original data back into this new array, and the original one released, we can now simply overwrite the old vertex array pointer, with that of the new array. We also increment the polygon's internal 'VertexCount' member variable to ensure that both the application and subsequent calls to this function have access to up to date information about the size of the vertex array.

```
    // Store the new buffer
    Vertices     = VertexBuffer;
```

```
    // Increment vertex count
    VertexCount += nVertexCount;
```

The final task that must be undertaken here is to return the index to the first of the vertex elements added to the array during that call. This index will be equal to the number of vertices that were originally stored in the polygon before this function was called. Returning the correct index can be achieved in one of two ways, either by making a temporary copy of the 'VertexCount' variable before we increment it, or by subtracting the requested additional vertex count parameter from the newly incremented 'VertexCount' value. Our implementation chooses the latter of these two methods as shown in the following snippet:

```
    // Return the base vertex
    return VertexCount - nVertexCount;
}
```

Although there is a relatively small amount of code in this function, it was important to discuss this given the changes in the way we are handling errors within our pre-processing tool. This exception handling scheme is used throughout this new application, and as a result we will not be going into quite as much detail about future memory allocation routines.

CPolygon::InsertVertex

There are relatively few cases in which we will ever need to insert a vertex into a polygon after it has been built. In most cases we will be generating new polygons with which we would build new vertex arrays from scratch each time. With that said, there are situations in which certain tasks are made easier by inserting a single vertex into a particular position around the polygons exterior. An example of such a case is with the recently covered T-Junction repair process. Because we have covered this function recently, we will only provide a brief recap to demonstrate this function as it applies to our new tool.

As before, this function's main purpose is to reshuffle the polygon's internal vertex array in order to create space for a new vertex. This vertex will be inserted *before* the vertex specified by the single parameter passed to this function. By doing so, we ensure that the new vertex will have an index that is equal to the value requested by the calling function.

```
long CPolygon::InsertVertex( unsigned long nVertexPos )
{
    CVertex *VertexBuffer = NULL;

    // Add a vertex to the end
    if ( AddVertices( 1 ) < 0 ) return -1;
```

The first thing we need to do here is to resize our vertex array to make sure that here is enough room for a single new vertex to be inserted. As outlined in our earlier coverage, the 'AddVertices' function will return a value of -1 if the allocation was unsuccessful. If this is found to be the case, then this function will return immediately using a similar error code.

At this stage, the polygon remains unaltered with the exception that there is a new empty vertex at the end of the internal vertex array. Due to the fact that our application requires a free slot to be created at an arbitrary position within the array – not necessarily at the end – we must shuffle the array elements such that this spare vertex now exists at the correct location. This is achieved by moving each of the vertices along by one, starting with the final element from the original vertex array, continuing backwards until we have moved each vertex up to and including the element specified by the input parameter. This could be achieved using the following code:

```
long i; // Signed type, we're counting backwards
for ( i = VertexCount - 2; i >= nVertexPos; ++i )
{
    // Move the current vertex forward by one.
    Vertices[i + 1] = Vertices[i];

} // Next Vertex
```

In this code you can see that we are simply assigning each vertex to the value of the one that falls directly prior to it in the vertex array. This is repeated until we reach the requested position, leaving a gap in the array at the element on which the loop exits.

Although this is a relatively simple loop, there is a quick and easy way to achieve this same effect using the 'memmove' function exported by the standard C runtime library.

```
    // Reshuffle the array unlesss we have inserted at the end
    if ( nVertexPos != VertexCount - 1 )
    {
        // Move all the verts after the insert location, up a ways.
        memmove( &Vertices[nVertexPos + 1], &Vertices[nVertexPos],
                 ((VertexCount-1) - nVertexPos) *  sizeof(CVertex) );

    } // End if not at end.
```

Internally, this function works in a similar fashion to the looping method described previously. The 'memmove' function is not however concerned with the layout and structure of the data or any form of class assignment or protection. It basically performs a byte for byte transfer of the specified area of memory, into the required destination address. This is not a problem for our vertices because they are essentially just data structures that do not use any form of virtual function mapping that may cause problems when moving the array contents in this way. One test that we must perform before we call this function however is to determine whether the requested vertex position is already at the end of the array. If this was found to be the case, then the data area that we would pass to this function would be empty and could potentially cause the operation to fail. Since the previous call to the 'AddVertices' function results in the new vertex being added to the end of the array, we need take no further action if this was the position requested by the calling function.

Once the contents of the array have been moved such that a gap remains at the correct location, we must finally overwrite the values of that spare vertex with those of a default state and return a success code to the calling function.

```
    // Initialize data to default values
```

```
    Vertices[ nVertexPos ] = CVertex( 0.0f, 0.0f, 0.0f );

    // Return the position
    return nVertexPos;
}
```

CPolygon::Split

The polygon split function remains largely unchanged from the implementations we have seen in earlier lab projects. However, there are a few alterations that have been made to allow us to take full advantage of our new hierarchical polygon class design.

```
HRESULT CPolygon::Split( const CPlane3& Plane, CPolygon * FrontSplit,
                         CPolygon * BackSplit, bool bReturnNoSplit )
{
    ...

    // Local variable definitions remain unchanged

    ...

    // New local variable
    CVertex NewVert;

    ...

    // Temporary array allocation and early-out logic is
    // largely unchanged. As a result it is not included here.

    ...
```

Although this function is largely identical to the previous implementation, there are some changes that affect how the ‘FrontSplit’ and ‘BackSplit’ parameters are declared and how they should be used by the application. Up until this point, the split function has been responsible for allocating any new polygon instances in cases where the source polygon is found to be spanning the input plane. These new polygon instances were returned to the calling function via the ‘FrontSplit’ and ‘BackSplit’ parameters which were previously declared as double ‘CPolygon’ pointers:

```
Split( const D3DXPLANE& Plane,
       CPolygon ** FrontSplit, CPolygon ** BackSplit, bool bReturnNoSplit )
```

Recall that our application would pass in a pointer to another ‘CPolygon’ pointer *variable* that would then be dereferenced by the split function in order to store the new polygon pointers in the variables referenced by these two parameters.

In this implementation we do not wan to force the application to use any particular polygon class at any point in the process. If the split function was to allocate any new polygon objects using the ‘CPolygon’ type, then we would be unable to make use of any form of class hierarchy in this way. As a result, the calling function must now be responsible for creating the polygon instances in advance – using the

applicable class type – and pass in these already instantiated objects via these same two parameters. Because we are no longer passing in pointers to variables as before, these two parameters are now simply single 'CPolygon' pointers. Although it may seem as though we are still placing a restriction on the application by accepting pointers of this base class type, remember that in C++ we are able to use a pointer to any type found within the class hierarchy when we need to access the superclass information. In the case of this base split function, we will only be modifying the properties defined by the base 'CPolygon' class itself.

We mentioned in the comments included in the previous code snippet that all of the logic we have previously implemented for the temporary array allocation and any 'early out' testing remains the same. Let us therefore move on to discussing those changes made to the core splitting functionality.

```
    // Compute the split if there are verts both in front and behind
    if (InFront && Behind)
    {
        for ( i = 0; i < VertexCount; i++)
        {
            // Store Current vertex remembering to MOD with number of vertices.
            CurrentVertex = (i+1) % VertexCount;

            if (PointLocation[i] == CLASSIFY_ONPLANE )
            {
                if (FrontList) FrontList[FrontCounter++] = Vertices[i];
                if (BackList)  BackList [BackCounter ++] = Vertices[i];
                continue; // Skip to next vertex
            } // End if On Plane

            if (PointLocation[i] == CLASSIFY_INFRONT )
            {
                if (FrontList) FrontList[FrontCounter++] = Vertices[i];
            }
            else
            {
                if (BackList) BackList[BackCounter++] = Vertices[i];

            } // End if In front or otherwise

            // If the next vertex is not causing us to span the plane then continue
            if ( PointLocation[CurrentVertex] == CLASSIFY_ONPLANE ||
                 PointLocation[CurrentVertex] == PointLocation[i]) continue;

            // Calculate the intersection point
            Plane.GetRayIntersect( Vertices[i], Vertices[CurrentVertex],
                                   NewVert, &fDelta );
```

The above code also remains largely unchanged. The only real difference here is that we are no longer using our collision library to retrieve information about the relative locations of each vertex in respect to the specified plane. Notice the last line in this snippet in which we are using one of our new 'CPlane3' class functions named 'GetRayIntersect'. This is in place of the previous call to the 'CCollision' class's 'RayIntersectPlane' function that has not been included in this application. Unlike the previous ray

intersection function – which required a ray start point and a ray velocity – this function accepts both a ray start and end point as its input parameters. Notice however that we are passing the vertices directly into this function without any sort of casting, even though the function we are calling is declared using 'CVector3' parameter types. This is just one of the benefits of deriving our vertex from the vector class as discussed earlier.

The 'GetRayIntersect' function outputs both the point in space at which the ray intersects the plane, as well as the optional 't' value we are familiar with. In the initial code listing for this function you may have noticed that we included a new local variable named 'NewVert'. This variable was also of the type 'CVertex'. Once again, because of our vertex class hierarchy design, the intersection function is able to set up the vertex values directly even though it is expecting a parameter of the type 'CVector3'. We also pass to this function the floating point variable 'fDelta', into which the intersection 't' value will be stored. We will need both of these values in order to correctly interpolate the remaining vertex components.

In this next block of code we perform these remaining vertex component interpolation steps. This ensures that after the polygon has been split, any new vertices inserted by the splitting procedure contain the correct values based on their position along the edge of the original polygon. Once the new vertex has been completely initialized, we then store it in both of the front and back fragment lists as before.

```
            // Interpolate Texture Coordinates
            CVector3 Delta;
            Delta.x    = Vertices[CurrentVertex].tu - Vertices[i].tu;
            Delta.y    = Vertices[CurrentVertex].tv - Vertices[i].tv;
            NewVert.tu = Vertices[i].tu + ( Delta.x * fDelta );
            NewVert.tv = Vertices[i].tv + ( Delta.y * fDelta );

            // Interpolate normal
            Delta          = Vertices[CurrentVertex].Normal - Vertices[i].Normal;
            NewVert.Normal = Vertices[i].Normal + (Delta * fDelta);
            NewVert.Normal.Normalize();

            // Store in both lists.
            if (BackList)  BackList[BackCounter++]   = NewVert;
            if (FrontList) FrontList[FrontCounter++] = NewVert;

        } // Next Vertex

    } // End if spanning
```

Once the front and back vertex lists have been created, we then store this data into the applicable polygon fragments allocated and passed in by the application. Unlike previous polygon class implementations however, this class does not store any additional information which needs to be duplicated into those fragments. This will in fact be handled by those classes which are derived from this base class. We will see how this achieved in the next section where we will discuss one of these derived classes.

```
    // Allocate front face
    if (FrontCounter && FrontSplit)
    {
```

```
        // Copy over the vertices into the new poly
        FrontSplit->AddVertices( FrontCounter );
        memcpy(FrontSplit->Vertices, FrontList, FrontCounter * sizeof(CVertex));

    } // End If

    // Allocate back face
    if (BackCounter && BackSplit)
    {
        // Copy over the vertices into the new poly
        BackSplit->AddVertices( BackCounter );
        memcpy(BackSplit->Vertices, BackList, BackCounter * sizeof(CVertex));

    } // End If
```

With our polygon fragments fully constructed, the only thing that remains for us to do is to clean up any of the temporary arrays that may have been allocated at the beginning of the function, and return our success code.

```
    // Clean up
    if (FrontList)      delete []FrontList;
    if (BackList)       delete []BackList;
    if (PointLocation)  delete []PointLocation;

    // Success!!
    return BC_OK;
}
```

CPolygon:: ReleaseVertices / ~CPolygon()

The final two base polygon member functions are those that are responsible for cleaning up and releasing any memory allocated and stored within this class. The first of these is the 'ReleaseVertices' function. This is a public member function that can be called by the application if it needs to release the memory allocated by this polygon, perhaps in order for this object instance to be reused. The second of these functions is the class destructor. Rather than duplicate the clean up code provided by the 'ReleaseVertices' function, the class destructor simply calls it at the point where the polygon object is released or goes out of scope.

```
void CPolygon::ReleaseVertices()
{
    // Clean up after ourselves
    if ( Vertices ) { delete []Vertices; Vertices = NULL; }
    VertexCount = 0;
}
```

```
CPolygon::~CPolygon()
{
    // Clean up after ourselves
    ReleaseVertices();
}
```

With our polygon base class defined, let us now take a look at how we might derive a new polygon class that will provide the data members needed to store any renderable polygon information loaded from file.

The CFace Class

This is the first of our derived polygon classes. It is used by a large portion of our application for the storage of renderable polygon information. That is the polygon data which is loaded from file, compiled and exported, and is finally intended to be rendered by our game application. As a result, this class stores the common pieces of information we have come to rely on such as the polygon normal, texture and material information in addition to any flags and alpha blending properties that may be required.

Recall during our discussion of the new polygon class hierarchy design, we ideally want to create a class structure in which each module is responsible for defining its own variables within an independent polygon class. However, at this point in time we need an intermediate concept that we can use to load the data contained within the source IWF file. This will enable us to initialize each processing module easily, using a common polygon structure, from which they can each base their own storage classes. At a later point in this lesson, we will derive a further class from this one which will be responsible for storing additional information required during the BSP compilation process for instance.

Due to the fact that this class is derived from the 'CPolygon' base class, much of the functionality is inherited. As a result, we are only really responsible for extending this class for the purposes of storing and retrieving any additional render based information. Let us therefore take a look at the declaration for this intermediate storage class.

```
class CFace : public CPolygon
{
public:
    // Constructors & Destructors
    CFace( );

    // Public Variables for This Class
    CVector3        Normal;                 // Face Normal
    short           TextureIndex;           // Index into texture look up
    short           MaterialIndex;          // Index into material look up
    short           ShaderIndex;            // Index into the shader look up
    ULONG           Flags;                  // Face Flags
    UCHAR           SrcBlendMode;           // Face Source Blend Mode
    UCHAR           DestBlendMode;          // Face Dest Blend Mode

    // Public Virtual Functions for This Class
    virtual HRESULT Split( const CPlane3& Plane, CFace * FrontSplit,
                           CFace * BackSplit, bool bReturnNoSplit = false );

};
```

Earlier we mentioned that any applicable member functions have been omitted from many of the class declarations being discussed throughout this chapter. In the case of this derived class however, with the exception of the virtual 'Split' function shown, this class defines no additional member functions. Much of the functionality available to the application through an instance of this type is provided by the base

'CPolygon' class from which it inherits. There are however several new member variables declared here that are discussed below.

CVector3 Normal
This member variable is responsible for storing the normal vector for this polygon / surface. As you should already be aware, this piece of information is extremely important to us when we are constructing a *polygon aligned* BSP tree compiler. This is due to the fact that both the polygon normal, and at least one of its vertices, is used as the basis for creating the separating planes assigned to each node in the tree.

short TextureIndex
Previously we mentioned that this class is responsible for storing each of the pieces of information that may be required to render this polygon in our final application. As a result, we store the index into a texture information array in this member variable. Although this application does not perform any sort of rendering itself, the texturing information must be exported back out to file when the compilation process has been completed. Notice that we declare this member variable using a signed type. This is because we use a value '-1' to signify that no texture has been assigned to this particular polygon.

short MaterialIndex
As with the aforementioned 'TextureIndex', we also store an index into the global material array much as we have done before in previous lab projects. Again this is a signed type in which '-1' signifies that no material has been applied to this polygon and a default material should be assumed. Although we have not yet discussed the application texture and material arrays at this point, we will see later how this information is loaded from file within our main compiler class.

short ShaderIndex
At this point in the course we have not yet encountered the use of pixel or vertex shaders within our application. However, we are currently constructing a tool which will hopefully be applicable to future lessons. As a result we also retain additional pieces of information that are available in the source IWF file such as the per-polygon shader index. As with each of the texture and material indices, a value of '-1' signifies that no shader has been applied to this polygon.

ULONG Flags
As we have discussed in previous lessons, each of the surfaces stored within the IWF file also carries with it a number of possible flags. These include as an example whether the polygon should be considered invisible or not. Although we do not pay an attention to this information within the various compilation processes in this application, the flags assigned to each polygon should be retained in order for them to be exported back out to the compiled file.

UCHAR SrcBlendMode
The source alpha blending mode is again another piece of information that is only really applicable to our rendering application. This information is loaded from file and retained here only for the purposes of exporting it back out to file for the run-time application's import procedure to process.

UCHAR DestBlendMode

This member variable is the complement of the source blending mode variable and is provided only for the purposes of export for the rendering application.

Although many of these member variables serve no other purpose than to be exported to the resulting file, it is important that this information is retained and is processed as each compiler task is performed. During the BSP compilation process for instance, we already know that many polygons will be discarded and likewise new polygons created in their place. As a result it is imperative this information be considered in several places in the application and – as we will discover shortly – one such place is during the splitting of our polygon data against the separating planes.

This class is of course merely an extension of the base 'CPolygon' class we covered in great detail in the previous section. As a result, there are only two member functions exposed by this class, each of which we will be discussing next.

CFace::CFace()

As with most of our classes, we start with the class constructor. We have defined several new member variables in this derived class that we need to initialize. Remember that in C++, the constructor of the base classes are called automatically when an object of this type is created. As a result we need only initialize the variables that are declared explicitly by this class declaration.

```
CFace::CFace()
{
    // Initialise anything we need
    MaterialIndex = -1;
    TextureIndex  = -1;
    ShaderIndex   = -1;
    Flags         =  0;
    SrcBlendMode  =  0;
    DestBlendMode =  0;
}
```

CFace::Split

Here is where our new hierarchical polygon class design comes into its own. Even though we have declared many new member variables that need to be duplicated into each of the split fragments, we need not re-implement the split functionality provided by the base 'CPolygon' class.

```
HRESULT CFace::Split( const CPlane3& Plane, CFace * FrontSplit,
                      CFace * BackSplit, bool bReturnNoSplit )
{
```

As before, the first parameter to this derived 'Split' function is a reference to the separating plane that will be used to classify and potentially split the polygon into two component fragments. The second and third parameters are pointers to each of the *pre-allocated* polygon instances into which the split function will place the output data. Notice however that the types of these two polygon pointers are declared as 'CFace', rather than 'CPolygon' as before. This allows us to both inform the application about the type of polygon this function is expecting, as well as granting us easy access to the newly defined member

variables within this class. This is why it was vital that we made the alterations discussed earlier, such that the application would pass in pointers to two pre-allocated polygon objects of the correct type. Because we are being passed two new 'CFace' objects via the 'FrontSplit' and 'BackSplit' parameters, we will be guaranteed to have access to the new member variables that we have declared.

The fourth and final parameter is the early out flag that informs the split procedure whether or not to return immediately if no split has occurred. Recall that if the application passes true to this parameter and the polygon's vertices are found not to span this plane, then the function will exit immediately without copying any information into either of the child fragments. If the application should specify false, then this function would proceed to copy the vertex data into the appropriate fragment even if no split has occurred.

Due to the fact that a large portion of the tasks undertaken during the splitting of a polygon are related to the building and separating of vertex data, there is obviously a lot of common functionality that we have already implemented in the base 'CPolygon'. As a direct result of deriving the 'CFace' class from 'CPolygon' in this way, we are able to make use of this functionality by simply calling the base class' 'Split' function, passing in each of the parameters specified by the application directly:

```
    // Call base class implementation
    HRESULT ErrCode = CPolygon::Split( Plane, FrontSplit, BackSplit,
                                       bReturnNoSplit );
    if (FAILED(ErrCode)) return ErrCode;
```

At this point, the base class implementation of the split function will have separated any applicable vertices into the appropriate polygon fragments specified by the application. Of course the base implementation is not aware of any of the member variables that have been included in this extended polygon class. To this end, the only remaining task that our derived class's split function must undertake is to copy the data stored within each of these variables, into those of the new split fragments as shown in the following snippet.

```
    // Copy remaining values
    if (FrontSplit)
    {
        FrontSplit->Normal        = Normal;
        FrontSplit->MaterialIndex = MaterialIndex;
        FrontSplit->TextureIndex  = TextureIndex;
        FrontSplit->ShaderIndex   = ShaderIndex;
        FrontSplit->Flags         = Flags;
        FrontSplit->SrcBlendMode  = SrcBlendMode;
        FrontSplit->DestBlendMode = DestBlendMode;
    } // End If

    if (BackSplit)
    {
        BackSplit->Normal         = Normal;
        BackSplit->MaterialIndex  = MaterialIndex;
        BackSplit->TextureIndex   = TextureIndex;
        BackSplit->ShaderIndex    = ShaderIndex;
        BackSplit->Flags          = Flags;
```

```
        BackSplit->SrcBlendMode   = SrcBlendMode;
        BackSplit->DestBlendMode  = DestBlendMode;
    } // End If

    // Success
    return BC_OK;
}
```

You can hopefully gather from this simple function that: in creating a relatively simple class hierarchy design, we are able to easily extend the functionality of the base class and provide new custom member variables at will. This will be of enormous benefit to each of the future compilation process modules that we will construct because it allows them to create their own polygon structures and store custom information without having to re-implement any of the common polygon functionality.

One other item of note is that we have not included a destructor in this class. Remember that because this class is derived from 'CPolygon', this base class destructor will always be called. Due to the fact that none of the member variables in the extended 'CFace' class contain pointers to any form of allocated memory we do not need to include any release or cleanup code. The vertices that were allocated will however be released by the base class constructor automatically, even if we were to include a destructor here.

With the 'CFace' class fully defined, we are now able to load and save the renderable polygon information to and from the source and destination IWF files. However, we still need to implement a class that serves as the container for this polygon data.

The CMesh Class

The mesh class is a concept that will already be very familiar to us. We have developed several different mesh classes in previous lab projects, but these too were heavily biased toward the construction of hardware friendly buffers used for rendering. As with the polygon classes we have covered so far, the mesh class we are developing in this tool is intended to store information only for the purposes of easy access to the scene data and final export to file.

```
class CMesh
{
public:
    // Constructors & Destructors
             CMesh( );
    virtual ~CMesh( );

    // Public Variables for This Class
    char           *Name;                // Stored name loaded from IWF
    CFace         **Faces;               // Mesh faces
    ULONG           FaceCount;           // Faces in this mesh
    CMatrix4        Matrix;              // Meshes object matrix
    CBounds3        Bounds;              // Meshes local space bounding box
    ULONG           Flags;               // Mesh flags (i.e. detail object)

    // Public Member Functions Omitted
```

```
};
```

This mesh class is relatively simple and declares only a handful of member variables. These variables match the information stored within the IWF file. Let us examine each of these in detail.

char * Name
Each of the meshes contained in the scene geometry file can have a name associated with them. This name string is usually specified by the artist or level designer within a scene editor such as GILES™ which can be used by the application to identify a particular mesh. As an example of why this might be useful, imagine that we created a mesh that served as a door within the level. Our application might want to open this door whenever the player triggered a script by entering the correct code on a door panel. Rather than opening every door in the level, this script would obviously need to identify a particular door to open. By naming each of the door meshes individually, the script writer would then be able to identify just the specific door that should open when the correct code has been entered into that panel.

Again this information is not used directly within our compilation tool, but we are required to retain this string in order to export it once again, after the compilation procedures have been executed.

CFace ** Faces
Because the mesh data stored in the file will ultimately be used for rendering by the final game application, this mesh class is designed to store and manage polygons using the 'CFace' class type. As a result, this member variable stores a pointer to an array of 'CFace' object instances rather than its base class 'CPolygon' found in previous implementations. Notice how we are using a double pointer declaration here. This is because we want to store an array of 'CFace' *pointers* which allows us to perform a simple shallow copy on the array each time it is resized. Just as our polygon class exposed a series of functions to manage the internal vertex array, the mesh class does the same in order to allow our application to manage this polygon array in an easy to use fashion.

ULONG FaceCount
This variable simply stores a value describing the number of polygons currently stored in the aforementioned 'Faces' array. We need to maintain this information in order to resize this array in each call to the 'AddFaces' member function, in addition to correctly reporting the length of the array to the application.

CMatrix4 Matrix
In previous applications, mesh object matrices were maintained by a wrapping 'CObject' class. This was provided in order to allow our application to create references to individual meshes and still maintain the ability to position each mesh independently. In the IWF file however, this reference information is maintained by individual 'entity' structures within the file and will be handled separately. In effect, each individual mesh may have its own matrix that describes the position and orientation of the *first* instance of that mesh. This first mesh may then be subsequently referenced by an entity that will override this position and orientation information. Again, because we are not rendering this mesh data, we have no need for the wrapping object class concept and therefore this matrix is simply maintained directly within the mesh itself. In GILES™, the vertices stored within each polygon are expressed in world space directly. This means that for the most part this member variable will contain an identity matrix. With that said, other scene creation tools may export an IWF file that does not define its mesh vertices in

world space. This imported variable should therefore still be taken into consideration. We will discuss how this matrix is used within the pre-processing tool a little later on in this lesson.

CBounds3 Bounds
This variable describes the axis aligned bounding box extents of the mesh vertex data in world space. The bounding box information provides us with a quick initial broad-phase mechanism by which we can test for intersection between two meshes. Among other things, this variable will be used later on in this chapter during the hidden surface removal process to determine whether two meshes might need to be merged together using the CSG union operation.

ULONG Flags
Just as with the polygon / surface information stored within the IWF file, each mesh also carries with it a number of possible flags. These include as an example whether the mesh itself is a detail object that should not be considered during the BSP compilation step. We will discuss the implications of these flags when they are encountered during the coverage of each of the main compilation tasks.

Let us now discuss the functions to the CMesh class that were omitted from the class declaration. We will start by looking at the initialization procedure (the constructor) and will then discuss the array management. Finally we will look at the various utility functions that allow us to more easily integrate the mesh into the compilation tool.

CMesh::CMesh()

The only constructor defined by the new mesh class is the default one, which simply initializes the member variables of the mesh whenever an instance of this class is created.

```
CMesh::CMesh()
{
    // Initialise anything we need
    Faces       = NULL;
    FaceCount   = 0;
    Flags       = 0;
    Name        = NULL;
}
```

As you can see, we simply set each of the member variables to a default state of either NULL or 0. In the case of the two pointer variables – ‘Faces’ and ‘Name’ – we set them to an initial state of NULL in order for us to identify whether or not any memory has already been allocated and should potentially be released. If either of these values is found to be equal to NULL at a later point, then we should obviously not attempt to release the memory to which this variable points because it had not yet been allocated.

CMesh::AddFaces

This is a public method that is used by the application in order to grow the ‘CFace’ array maintained by an instance of this mesh class by the specified amount. Lab project 16.2 uses this function when loading internal meshes within the IWF file in addition to cases where it is required to manually construct a new mesh as we will see shortly.

The ‘AddFaces’ function is similar in many respects to the memory allocation functions we have been using throughout the development of each lab project in this series. However, as with the polygon’s ‘AddVertices’ function, there are a number of key differences that we must draw your attention to.

```
long CMesh::AddFaces( unsigned long nFaceCount )
{
    CFace **FaceBuffer = NULL;

    // Validate Requirements
    if ( nFaceCount == 0 ) return -1;
```

The only parameter accepted by this function allows the application to specify how many additional polygons we would like to add to the ‘CFace’ array stored here. If the array has already been created by a previous call, this will be the number of elements by which that array will be grown. As in the many functions we have implemented before, this parameter is validated to ensure that the allocation will not fail due to the application specifying a count of zero.
This function has a return type of a simple signed long. This will be used to return the index to the first of the polygon elements added to the array during that call. This index – as always – will be equal to the number of polygons that were originally stored in this mesh before the ‘AddFaces’ function was called. If an error occurred during the execution of this function, a value of ‘-1’ will be returned.

As before, our first job is to attempt to allocate a new array of the same type as the internal ‘Faces’ member variable that has been correctly sized to take into account those new requested polygons. Subsequently, any data already stored within the mesh must then be copied into this new polygon array. Because we are simply storing *pointers* to heap allocated ‘CFace’ objects, this can be achieved by a single call to the standard C runtime ‘memcpy’ call. This new array will eventually be used to overwrite the one currently stored by the mesh.

```
    try
    {
        // Allocate brand new buffer
        FaceBuffer = new CFace*[ FaceCount + nFaceCount ];
        if (!FaceBuffer) throw std::bad_alloc(); // VC++ Compat
        ZeroMemory( FaceBuffer, (FaceCount + nFaceCount) * sizeof(CFace*) );

        // Copy over any old data
        if (Faces) memcpy( FaceBuffer, Faces, FaceCount * sizeof(CFace*));

        // Allocate new faces
        for ( long i = 0; i < (signed)nFaceCount; i++ )
        {
            FaceBuffer[FaceCount + i] = new CFace;
            if (!FaceBuffer[FaceCount + i]) throw std::bad_alloc(); // VC++ Compat

        } // Next face

    } // End Try Block
```

There is one very important addition that has been made to the above code that we have not yet encountered in any of our memory allocation routines. The latter part of the function actually allocates the new 'CFace' objects on behalf of the application and stores them in the array in advance. There are several reasons why we have chosen to do this with our new mesh class. The first of these is due to our new class hierarchy design. Because the application has much more freedom to create an object using one of the several potential types of polygon classes, we have introduced a situation in which the application could create a polygon object of a type which does not store much of the information required. This could also lead to a situation in which one of the virtual functions defined by the polygon class could be called erroneously. By allocating the polygon objects explicitly within this function, we are able to enforce the type of polygon stored within each element of the mesh's member array. An additional benefit of including this allocation step here is that we are able to make use of the error handling logic in this function without having to include that same logic each time we need to allocate a mesh polygon.

At this point in the function, we have allocated the newly sized array and copied over any of the polygon pointers previously stored in the 'Faces' member variable. A series of new 'CFace' class instances have also been created and initialized by their own constructors, and finally stored within this new array. As always, there are potentially many situations under which an error may have occurred with all of the memory allocation that has been performed.

All of the allocation steps within this function have been wrapped in a try block that will allow us to capture any exceptions that may have been thrown during the execution of the lines of code within that block. Backwards compatibility has also been preserved by testing the pointer value returned from each allocation operation and explicitly throwing a similar exception if the value was found to be 'NULL. Of course, an exception may be thrown at any point in this block of code that might result in only some of the intended allocations having been attempted. Therefore, in the event of an exception, we must clean up and release those memory areas that were successfully allocated. In the following catch block, a great deal of care is taken to test each of the potentially allocated memory pointers and release them only if they are *not* equal to NULL. This should ensure that we do not attempt to deallocate memory that was not allocated prior to the exception being thrown.

```
    // Did an exception get thrown ?
    catch (...)
    {
        if (FaceBuffer)
        {
            // Release any already allocated faces
            for ( long i = 0; i < (signed)nFaceCount; i++ )
            {
                if (FaceBuffer[FaceCount + i]) delete FaceBuffer[FaceCount + i];

            } // Next face

            // Release buffer
            delete []FaceBuffer;

        } // End if FaceBuffer

        return -1;
```

```
    } // End Catch Block
```

With our correctly sized array now allocated and initialized with new 'CFace' object pointers, the next job is of course to release the old 'Faces' array, and overwrite the member variable with the new heap allocated array pointer. Once this is done, we can increment the internal 'FaceCount' variable to ensure that we keep an accurate record of the number of polygons stored in the polygon array.

```
    // Free up old buffer and store new
    if (Faces) delete []Faces;
    Faces = FaceBuffer;

    // Increment face count
    FaceCount += nFaceCount;
```

Finally, with the task of adding new faces complete, this function returns the array element index to the first polygon that has been added. This allows the application to retrieve and iterate through the polygon array beginning with the correct element.

```
    // Return the base face
    return FaceCount - nFaceCount;
}
```

CMesh::BuildFromBSPTree

Although we have not yet discussed our new BSP leaf tree class, we should be familiar enough with the general concepts involved in the construction of leaf trees to know that, once the compilation has completed, the tree will store a list of each of the polygons contained within its leaves. Once the BSP tree has been built, we need to write the polygon information to file in a format that is easy for the application to retrieve. Due to the fact that our application is already capable of loading mesh and polygon data from the IWF file, it makes sense that we export the resulting BSP tree polygon data out to file as a standard mesh. This will prevent us from having to implement another series of polygon import functionality and will allow the BSP tree's polygon data to be loaded seamlessly by *any* IWF import procedure.

This 'BuildFromBSPTree' function has been provided in order to help us convert the BSP tree polygon data into a format compatible with the pre-defined IWF mesh / polygon structures. It can also be used by any additional compilation process in situations where the polygon information must be extracted. We will discuss the process by which the remainder of the BSP tree information is accessed and exported at a later stage, but for now this is a good example of how a mesh might be constructed using existing polygon information.

```
bool CMesh::BuildFromBSPTree( const CBSPTree * pTree, bool Reset /* = false */ )
{
    ULONG Counter = 0, i;

    // Validate Data
    if (!pTree) return false;
```

There are two parameters declared by this function. The first is the BSP tree class instance from which we would like to extract the polygon information. We need not concern ourselves with how this class is defined at this point because our primary focus is on how the mesh is being constructed.

The second 'Reset' parameter allows the application to specify whether or not the polygons contained within the specified BSP tree should be released before the specified BSP tree is processed. This parameter is optional, and has a default value of 'false'. If the application specifies 'true' to this parameter, then the mesh will be reset to a default state before extracting the polygons from the BSP tree and storing them in a newly built polygon array. Specifying 'false' will *prevent* the mesh from being reset. This will result in each of the BSP tree polygons being appended to the already existing mesh polygon array.

The very first task is to test the value of the 'Reset' parameter and reset our mesh to a default state if it found to contain a value of 'true'. This is achieved simply by making a call to the mesh's 'ReleaseFaces' function that is responsible for releasing all of the memory associated with the polygon array and resetting each of the associated member variables to their initial state. This function will be covered shortly. We must also reset the matrix stored within the mesh to an identity state.

```
    // Reset if requested
    if (Reset) { ReleaseFaces(); Matrix.Identity(); }
```

With the mesh having been reset or otherwise left intact, we next begin the task of building the mesh data. Before we do so however we need to determine how many polygons must be allocated in the mesh for the storage of those BSP tree polygons that we are interested in. Although we could simply allocate enough storage space for every polygon within the BSP tree and copy the data regardless of its status, there may be cases in which a polygon in the tree has been marked as deleted but has not physically been removed from the array. In addition, there may be polygons stored in the BSP tree that are invalid, however unlikely this may be. For this reason, it is advisable for this function to validate each of the polygons contained with the BSP tree and count the number of polygons that we will ultimately store.

This is demonstrated in the following code snippet in which we make use of two of the access functions exposed by the 'CBSPTree' class. These functions are named 'GetFaceCount' and 'GetFace' respectively. The 'GetFaceCount' function is responsible for retrieving the number of polygons / faces currently stored within the BSP tree class instance. Likewise, the 'GetFace' function is used to retrieve a pointer to a polygon situated at the specified index in the BSP tree's internal polygon array. This polygon is of a type derived from the 'CFace' class named 'CBSPFace' that is intended to extend our renderable polygon with additional BSP specific information. We will cover this additional polygon class in further detail a little later, but for now it should be fairly obvious as to the purpose of each of these additional variables.

The following code implements this validation technique. Put simply, this code loops through each of the polygons contained within the BSP tree and validates two of its exposed parameters. The first is the 'Deleted' member. In certain situations, such as during the hidden and exterior surface removal processes, polygons may need to be removed. Rather than physically remove the polygons from the tree, they are flagged as deleted. This allows us to undo a split operation if necessary, as we shall see in our coverage of HSR split repair later in this chapter. The second test we perform is simply to validate that this particular polygon contains at least the 3 vertices required to constitute a single triangle. If both of

these tests are passed, then a local counter variable is incremented. Once this loop has completed, the counter variable will contain the total number of polygons that we need to allocate in order to store every polygon that passed the validation tests.

With this tallying process completed, we then attempt to perform the actual allocation of the mesh's internal polygon array via a call to 'AddFaces'. Recall that the 'AddFaces' function returns either '-1' if a failure occurred, or the index to the first polygon added to the internal array should the allocation be successful. We will need this information a little later, so the return value from this function is retrieved and stored. Since the value currently stored in the counter variable we had previously used is no longer required, we can reuse this variable and overwrite its value with that returned by 'AddFaces'.

```
    // Count valid BSP tree faces
    for ( i = 0; i < pTree->GetFaceCount(); i++ )
    {
        if ( !pTree->GetFace(i)->Deleted &&
              pTree->GetFace(i)->VertexCount >= 3 ) Counter++;

    } // Next BSP Face

    // Allocate our faces
    Counter = AddFaces( Counter );
    if ( Counter < 0 ) return false;
```

At this point we have allocated the memory required to store the information for each of the BSP tree polygons that passed the initial validation tests. As a result we are now able to begin copying this polygon data into the mesh's internal array. We begin by looping through each of the BSP tree polygons once again, skipping over any of those which are considered to be invalid using the same two tests as were used in the previous loop.

```
    // Now we loop through and copy over the data
    for ( i = 0; i < pTree->GetFaceCount(); i++ )
    {
        CBSPFace * pTreeFace = pTree->GetFace(i);

        // Validate tree face
        if ( pTreeFace->Deleted ) continue;
        if ( pTreeFace->VertexCount < 3 ) continue;
```

With the validation tests carried out, we are guaranteed to have access to a valid BSP tree polygon at this point. Because of the fact that the BSP tree's 'CBSPFace' class is derived from the 'CFace' class, it might seem that we could simply copy over the pointer to this object without any adverse affects. However, we should not make any assumptions about the ownership of the BSP tree polygon data because there is no way for us to know if the application has further use for this information. If we were to simply copy the BSP tree polygon pointers into the mesh, and were to subsequently release that mesh, the BSP tree's internal array would now contain pointers to an invalid memory location. It is much safer and cleaner for us to perform a 'deep' copy at this point.

The following code does just this. Each of the 'CFace' members are populated using their counterparts in the 'CBSPFace' class retrieved from the BSP tree. Likewise, we also allocated a new vertex array for this new mesh polygon and duplicate the vertex data.

```
        // Retrieve faces for processing
        CFace    * pFace     = Faces[Counter];

        // Copy over data
        pFace->MaterialIndex = pTreeFace->MaterialIndex;
        pFace->TextureIndex  = pTreeFace->TextureIndex;
        pFace->ShaderIndex   = pTreeFace->ShaderIndex;
        pFace->Flags         = pTreeFace->Flags;
        pFace->SrcBlendMode  = pTreeFace->SrcBlendMode;
        pFace->DestBlendMode = pTreeFace->DestBlendMode;
        pFace->Normal        = pTreeFace->Normal;

        // Copy over vertices
        if ( pFace->AddVertices( pTreeFace->VertexCount ) < 0 ) return false;
        memcpy( pFace->Vertices, pTreeFace->Vertices,
                pFace->VertexCount * sizeof(CVertex) );

        // We used a new face
        Counter++;

    } // Next BSP Face
```

Note in the previous code that we used the 'Counter' variable to index into the mesh's newly allocated internal polygon array. This polygon would be the destination object into which this particular BSP polygon data would be copied. Recall that this variable was used to store the value returned from the 'AddFaces' function which specified the first polygon in the newly allocated set. Each time we find a valid polygon within the BSP tree and subsequently store it within the mesh, this counter variable must be incremented in order to move on to the next destination polygon.

The final task in this function, before returning a success code, is to update the mesh's bounding box information to take into account any new polygons that have been added to the mesh. This is achieved with a call to the 'CalculateBoundingBox' member function.

```
    // Calculate our bounding box
    CalculateBoundingBox();

    // Success
    return true;
}
```

Although it may not be clear at this point where and when we might make use of this function, it remains a useful discussion because it allowed us to examine a common usage of the mesh class. As mentioned earlier, we will be utilizing this function later on in the development of our compilation tool at which point any remaining questions you may have about this functions use should be resolved.

CMesh::CalculateBoundingBox

By now you should be very familiar with the techniques involved in calculating a bounding box using mesh polygon and vertex information. This is something we have had to implement many times, in several different parts of our previous run-time applications. Thankfully, the new 'CBounds3' utility class that we have implemented in this lab project performs the majority of the laborious work on our behalf.

The 'CalculateBoundingBox' function accepts no parameters. It simply iterates through each of the polygons contained within the mesh and passes the vertex data into the 'CalculateFromPolygon' bounding box member function. This function accepts a pointer to a set of vertex data, the number of vertices in that set, the size of the vertex structure used – which enables it to step to the positional information of each vertex in the array – and a final 'Reset' parameter that dictates whether to calculate new bounding box extent vectors from only that polygon, or to grow those extent values already contained within the object. In the implementation of this function we need to grow the bounding box for each polygon that is contained within the mesh. To this end, we reset the bounding box in advance and then specify 'false' to the reset parameter of the bounding box's member function.

Finally, once the mesh's 'Bounds' member has been calculated, this function returns to the calling function with a reference to this variable. This allows the application to use the result of the operation performed by this function as the input to another, without having to access the bounds member separately. Because this is achieved via the function return value, this is of course optional and the calling function need not consider this result.

```
const CBounds3& CMesh::CalculateBoundingBox()
{
    // Reset bounding box
    Bounds.Reset();

    // Loop through each face
    for ( long i = 0; i < (signed)FaceCount; i++ )
    {
        Bounds.CalculateFromPolygon( Faces[i]->Vertices, Faces[i]->VertexCount,
                                     sizeof(CVertex), false );

    } // Next face

    // Return our bounds reference
    return Bounds;
}
```

CMesh::ReleaseFaces / ~CMesh()

The final two functions in this class are those responsible for cleaning up and releasing any resources allocated by the mesh object. The first is a public method named 'ReleaseFaces'. This function is responsible for releasing only that memory allocated for the internal polygon array in addition to each of the heap allocated polygon objects stored within that array. As always, each value is tested to see whether it contains a value other than NULL. If this found to be the case then that object or array can be safely released. While this function can be called by the application at any point if it wishes to reuse a particular mesh object, this function is also called by the second of these two functions – the class

destructor – as the object is released or goes out of scope. In addition to calling the 'ReleaseFaces' function, the class destructor also releases any memory allocated for the storage of the 'Name' member string.

```
void CMesh::ReleaseFaces()
{
    // Clean up after ourselves
    if ( Faces )
    {
        // Release all face pointers
        for ( long i = 0; i < (signed)FaceCount; i++ )
        {
            if ( Faces[i] ) delete Faces[i];

        } // Next face

        // Release the array itself
        delete []Faces;
        Faces = NULL;

    } // End if Faces

    FaceCount = 0;
}
```

```
CMesh::~CMesh()
{
    // Clean up after ourselves
    ReleaseFaces();
    if ( Name ) delete[] Name;
}
```

Prerequisite Class Conclusion

At this stage we have now covered each of the prerequisite classes and functionality that we will need to have at our disposal as we move forward with the development of our new compilation tool. Although it may seem as though we have been covering a lot of topics that we have already discussed in previous chapters, there are many nuances to this new set of utility classes that we had to be aware of both in their implementation and their usage. Even though we have not yet begun to discuss the core implementation of the new pre-processing application, we have built a solid framework on which we can build that will make the job of creating both the tool itself, and each of the compiler modules a much more simple task.

With this new arsenal of utility classes at our disposal, let us now move on to the discussion of the application core.

Application Front End – Main.cpp & LogOutput.cpp

We discussed previously that we wanted to build a modular and portable tool that would enable us to integrate the core compiler functionality with any front end application that we might design. This could be a standalone GUI tool that serves only as the container for the compiler itself, or it could be another more comprehensive tool such as a level editor or modelling package. In lab project 16.2, we will be constructing a simple console like application that will serve as the front end for the compiler functionality. This will ensure that the code required to provide the interface does not obfuscate the more important compiler logic.

If we open and examine the source modules contained in the 'Application Source' folder of our source project, it is clear that the actual wrapper application that we will construct really is nothing more than the logic required to open a console window and output progress and log information to the end user. Given that the logging procedures constitute such a large portion of this lab project's responsibilities, let us quickly summarize how this process works.

Earlier in this chapter we touched briefly upon the mechanism by which the compiler integrates itself into the application for the purposes of writing progress and status information. Recall that this is achieved by using a base interface from which the application must derive. This base interface is named 'ILogger' and is declared in the header file 'Common.h' which can be found in the 'Support Source' project directory. This pure abstract interface class declares several virtual functions which must be implemented by the application derived class.

The following list outlines the purpose of each of these interface functions.

```
void LogWrite( unsigned long Channel, unsigned long Flags, bool NewMessage,
               LPCTSTR Format, ... );
```

This function is intended to be called by any part of the application to output messages to the user.

As a high level concept, the logging class is required to keep track of the information being output by several sources. These are referred to as channels within this implementation. Put simply, the application may be required to output the information for various different compiler process tasks that may be executed. These would be in addition to the general process status information that our application may need to display. Each compilation process is assigned a unique channel index that only that specific process will ever pass to the logging class member functions. The first parameter to the 'LogWrite' interface function is this channel index used to signify the destination to which this message is intended.

The second parameter can be passed one or more of the following flags:

```
#define LOGF_WARNING     1      // Log Flags : Display with warning format
#define LOGF_ERROR       2      // Log Flags : Display with error format
#define LOGF_UNDERLINE   4      // Log Flags : Display with underline property
#define LOGF_BOLD        8      // Log Flags : Display with bold property
#define LOGF_ITALIC      16     // Log Flags : Display with italic property
```

The derived class is not required to obey these flags due to the fact that in certain circumstances, such as in a console application, not all of these flags are even able to be supported. As a result, when calling the 'LogWrite' function, the values we specify should be considered to be only hints.

The third parameter, 'NewMessage' is used to inform the logging class whether or not this is the start of a new message. If a value of false is passed to this parameter then the logging class should consider this message to be a continuation of the previous.

The fourth and final defined parameter is the 'Format' value. You may have noticed that this method is declared using an ellipsis modifier. This format parameter is intended to be used in the same way as we would use the format parameter in the 'printf' function. If you are developing a derived logging class of your own, it is recommended that you look up the 'va_start', 'va_arg', 'va_end' and 'vsprintf' C++ runtime functions that can be used to interpret the format value in conjunction with variable argument lists.

```
void SetRewindMarker( unsigned long Channel );
void Rewind         ( unsigned long Channel );
```

This function is intended to cause the logging class to store a value indicating the current position within the specified channel when this function is called. This is used in conjunction with the 'Rewind' function to have the logging class roll back the channel text to the marker location recorded earlier. This is incredibly useful in tool environments like this where we might want to constantly update a piece of text within the log text with a percentage progress counter for instance.

```
void Clear( unsigned long Channel );
```

The derived class should clear the specified log channel of all text and marker information whenever this function is called.

```
ULONG GetCurrentChannel( );
```

Whenever a call is made to the 'LogWrite' function, the derived log class should record the most recently specified channel index. This value can then be retrieved by the application to continue writing to the same log channel as it wrote some initial text for instance.

```
void SetProgressRange( long Maximum );
void SetProgressValue( long Value );
```

The 'SetProgressRange' function is the means by which the application informs the derived logging class of the maximum value it might pass to the 'SetProgressValue' function. As an example, we might specify a value to the 'SetProgressRange' function of 900. With this value as a maximum, when we then pass a value of 450 to the 'SetProgressValue' function, the logging class would interpret this value to mean '50%' (i.e. 450/900 = 0.5 * 100 = 50).

```
void UpdateProgress( long Amount = 1 );
```

The 'UpdateProgress' function is the means by which the application has the logging class print the current progress percentage value to the current logging channel. Optionally, this function can be passed

a value used to increment the progress value as if it was passed in to the ‘SetProgressValue’ function. As an example, we might have set our maximum progress range to be equal to the number of polygons that we want to process. We might then call the UpdateProgress function, passing an amount value of ‘1’, each time we processed another polygon. In this case the logging class would automatically output a value of ‘100%’ at the moment we processed the final polygon and called ’UpdateProgess’ for the final time.

```
void ProgressSuccess( unsigned long Channel );
void ProgressFailure( unsigned long Channel );
```

These two functions are utility functions that can be called to have a success or failure message output to the specified channel in a format decided by the derived logging class developer. This might also allow the logging class to take further action when a success or failure occurred such as sending an e-mail to a technician should any process fail.

We will not go into any further detail about how we have implemented the derived console output class in this workbook. If you would like to take a look at the source for this class you can do so in the ‘LogOutput.cpp’ file in the ‘Application Source’ project directory.

With these general application concepts out of the way, lets move straight on to the application entry point function.

WinMain

Although we are constructing a console *like* application in this lab project, we have used a standard Win32 project to enable us to gain access to all of the benefits provided by such an application. This includes the ability to create and use the Windows™ common dialogs such as the standard ‘Open’ and ‘Save’ dialogs. In addition to supporting common Win32 API functionality, building an application of this type allows for future Win32-centric enhancements an example of which might be the inclusion of an additional window in order to preview the compiled scene perhaps.

Because we are developing our front end as a Win32 application, our entry point is defined using the standard ‘WinMain’ function, as opposed to the ‘main’ function we would use if we were creating a native console application. You should already be familiar with the WinMain entry point function at this stage, so we will not go into any further detail about the parameters or their intended use.

```
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine,
                    int iCmdShow )
{
    OPENFILENAME  File;
    TCHAR         FileName[MAX_PATH];
    CLogOutput    LogOutput;
    CCompiler     Compiler;
    bool          bSuccess;
    ULONG         i;
```

Notice in the above code that we have defined several local variables for use within this function. By far the most important of these is the ‘Compiler’ variable of the type ‘CCompiler’. This class is the one that

provides our application with its entire scene processing ability. We will be discussing this class in the next section of this chapter, but for now let us examine how the application might make use of the functionality exposed by this compiler class.

Due to the fact that this is not a native console application, the very first thing that our application must do is to open a console window that will serve as the channel through which the user is informed about the progress and status of each of the compiler tasks to be executed. This is achieved via a call to the Win32 'AllocConsole' function. This function will create the console window for use by our application but will *not* redirect the standard output to this window automatically. Functions and template classes such as 'printf' and 'cout' are designed to output string data through the 'stdout' data stream also know as the standard output stream or channel. In a native console application this data stream is automatically attached to the console that is implicitly created when the application begins. This would result in any string data passed to a call to 'printf' being output to the console window as we would expect. In a Win32 application however, this is obviously not the case because no console window exists without our explicit instruction. Therefore, we must also perform this operation ourselves by attaching the 'stdout' stream to the newly created console window once it has been created.

In C++, the standard output stream is defined by a global variable most easily accessed using the name 'stdout'. This is actually a variable of the type 'FILE' which is a structure provided by the standard C runtime library in the 'stdio.h' header file. You should already be familiar with this structure because a pointer of this type is returned and passed to functions we have used extensively in the past such as 'fopen' and 'fread'. Despite the name of this structure, the 'FILE' type can apply to any sort of data stream be that a literal file, or other data buffers such as that maintained by a simple console window. So, given the fact that 'printf' writes the string data out to the 'stdout' *file* we can take advantage of this fact by making a call to the 'freopen' function that allows us to redirect the output of an already existing data stream – such as 'stdout' – to a new destination. This function works in a similar fashion to the 'fopen' function in that it accepts a filename and a mode used to describe how the stream should be accessed. However, it also accepts a pointer to an existing 'FILE' object that we would like to redirect.

Obviously we do not want to save the data written to the standard output stream to a file and yet the first parameter of this function is expecting a filename. So how exactly does this function help us place the output of calls such as 'printf' to a console window? We can in fact utilize a feature of the underlying operating system to enable us to output string data to the console by specifying a string of "CONOUT$" to the path parameter of this function. This is essentially a special path name that is reserved for just this type of situation. In fact, we can use this path name with many of the file handling functions in order to access and output data to and from the console window. In order to integrate the standard output functions with the console window we must also pass "a" (or append) to the mode parameter, and finally the 'stdout' stream pointer variable to the final parameter.

With the 'stdout' *file* stream now redirected to the new console window, we should find that any function or library designed to write information to the standard output stream will now seamlessly output to our application's console window.

```
    // Create a console window so that we can use it as our logging output
    AllocConsole();

    // Redirect all standard output to console (i.e. printf, cout etc).
```

```
    freopen("CONOUT$", "a", stdout);
```

Now that the main window into our application has been created and attached, we can now begin to output logging and progress information to the user. The following code first initializes the instance to our custom 'CLogOutput' class by calling the 'Create' function. We pass the total number of channels that we would like it to maintain to this function. We use an arbitrary figure of 20 in this code to ensure that there are enough to be going on with.

Once we have initialized the logging class, we then output the leading messages for this application such as its name, version and copyright information. Refer back to our discussion of the base interface 'LogWrite' function in order to understand the meanings of the specified parameters.

Finally, we pass a pointer to the instance of our console logging CLogOutput class into the compiler. This will then be distributed throughout the system to allow the various components to output progress and status information to the user.

```
    // Create the log output handler and attach it to the compiler
    LogOutput.Create( 20 );
    LogOutput.LogWrite( LOG_GENERAL, 0, false,
                        _T("\nSolid Leaf BSP Tree Compiler v1.0.0\n"));
    LogOutput.LogWrite( LOG_GENERAL, 0, false,
                        _T("Copyright © 2005 GameInstitute.com.")
                        _T("All Rights Reserved.\n"));
    LogOutput.LogWrite( LOG_GENERAL, 0, true,
                        _T("Compiler startup successful, awaiting user input."));
    Compiler.SetLogger( &LogOutput );
```

The only piece of information that we need from the user in this application is the filename of the IWF file that they wish to be imported and passed through the various compilation procedures. In this application, rather than using a potentially cumbersome command line parameter scheme, this is achieved by using the Win32 'GetOpenFileName' function and associated structures that have been used and discussed in previous lessons. If the user selected a valid file then the 'GetOpenFileName' function will return a value of true, after storing the selected path and filename information in the variable specified through the 'lpstrFile' member of the 'OPENFILENAME' structure passed to this function. In our implementation this would be the local string variable named 'FileName'. If the user chose to cancel the open dialog, or some other failure occurred, then this function will return false and our application will exit.

```
    // Fill out our default file dialog structure
    ZeroMemory( &File, sizeof(OPENFILENAME) );
    ZeroMemory( FileName, MAX_PATH * sizeof(TCHAR));
    File.lStructSize  = sizeof(OPENFILENAME);
    File.hwndOwner    = NULL;
    File.nFilterIndex = 1;
    File.lpstrTitle   = _T("Select Input IWF File for Compilation.");
    File.Flags        = OFN_EXPLORER | OFN_HIDEREADONLY | OFN_FILEMUSTEXIST |
                        OFN_NOCHANGEDIR | OFN_PATHMUSTEXIST;
    File.lpstrFile    = FileName;
    File.nMaxFile     = MAX_PATH - 1;
```

```
    File.lpstrFilter  = _T("All Supported Formats\0")
                        _T("*.IWF;*.BWF\0")
                        _T("Interchangable World Format (*.iwf)\0")
                        _T("*.IWF\0");

    // Retrieve a filename.
    if ( !GetOpenFileName( &File ) ) return 0;
```

Once the user has selected the file that they would like to be processed, the application can then inform the compiler class of the user's selection via a call to the 'SetFile' function and begin the compilation process with a call to 'CompileScene'. It is important that we store the return value from the latter of these two functions because it will be used later to determine if the compilation process was successful or not.

```
    // Inform the compiler about the file to compile
    Compiler.SetFile( FileName );

    // Compile the scene
    bSuccess = Compiler.CompileScene();
```

In this application, these two lines of code constitute the majority of the application's interaction with the core of our compiler. Although it is possible to play a much more active role in deciding which processes the user would like to perform on the scene geometry, in addition to setting several available options for each processor module, we will use the default settings in this application. These default settings specify that every available compiler task should be enabled and executed using sensible process parameters. We will see the means by which we can construct and specify these options in the next section when we cover the actual compiler classes shortly.

With the compilation process completed, our application then proceeds to clear the console window using one of the handy member functions exposed by our console logging class – 'ClearConsole' -- and then requests that all of the logged process information for each channel is output to the console as a last step. This is achieved via a call to the 'CLogOutput::SwitchChannel' channel function. This allows the user to see the final results of each task undertaken by the compiler. If a failure was encountered, the user will be able to identify the step at which this failure occurred and potentially remedy that situation.

```
    // Clear the console window and output all of the status text one last time
    LogOutput.ClearConsole();
    for ( i = 0; i < 20; ++i )
    {
        // Was there any text in this channel?
        if (LogOutput.GetChannelText( i ) != NULL)
        {
            // Print the channel information out
            LogOutput.SwitchChannel( i, false );
            printf( _T("\n\n-------------------------------------------\n" );

        } // End if anything in channel

    } // Next Channel
```

If the compilation process was a complete success then the return code we previously stored in the local Boolean variable ‘bSuccess’ will contain a value of true at this stage. If this is found to be the case then we can instruct the compiler to save the resulting data to a new file as specified by the user. The next task therefore is to request that the user input a location and filename into which the compiled scene information will be exported.

Because we had previously initialized the ‘OPENFILENAME’ structure that was passed into the earlier call to the Win32 open dialog function, we need only alter those structure variables that must change in order to display the common *save* dialog with a call to the Win32 ‘GetSaveFileName’ function. If the user selected a valid output path and file name within this dialog, then the ‘GetSaveFileName’ function will return true, after having copied the selected file and path name into the local variable ‘FileName’ as before. This file name string can then be passed into the compiler object’s ‘SaveScene’ method in order to have the newly compiled scene data exported to that location.

If the compiler failed for any reason, or the user chose to cancel the save dialog, then the scene export request will be bypassed altogether as we move onto the closing stages of this function and our application front end.

```
    // Save the scene if it compiled succesfully
    if ( bSuccess )
    {
        // Alter structure for any changed options
        File.nFilterIndex = 1;
        File.lpstrTitle   = _T("Select Output BWF File.");
        File.lpstrFilter  = _T("BSP Compiled IWF (*.bwf)\0*.BWF\0");
        ZeroMemory( FileName, MAX_PATH * sizeof(TCHAR));

        // Retrieve a filename.
        if ( GetSaveFileName( &File ) )
        {
            // Save the scene
            Compiler.SaveScene( FileName );

        } // End if Selected File
        else
        {
            // Output information
            printf( "User chose to cancel save operation." );

        } // End if no file

    } // End if success
```

One of the downsides of running a console application in a windows environment is that as soon as the application exits, the console window is automatically closed and destroyed. It is often the case however, that the application may have output several pieces of important information to the console that the user may not see if the window were to close as soon as the process was complete. For this reason we include a call to the runtime ‘_getch’ console function which is designed to retrieve character input from the keyboard in a synchronous fashion. This means that the ‘_getch’ function will cause our application to wait until a key has been pressed before program control is returned and the remainder of

the code is executed. When combined with a standard ‘Press any key’ message, this will allow the user to examine any messages output to the console window at their leisure.

```
    // Hold application until user presses key
    printf( "\n\nPress any key to exit..." );
    _getch();
```

The final piece of code within this function simply releases the console window that we allocated earlier, and returns from the WinMain function with an exit code of 0. This will signal the end of the application’s lifetime and will then exit.

```
    // Clean up console
    FreeConsole();

    // Return
    return 0;
}
```

With our application front end fully implemented, let us now focus on the inner workings of our pre-processing application by examining the core compiler classes.

The Compiler Core – CCompiler.cpp

The core of the compiler tool we are developing in this lesson is centered on the ‘CCollision’ class found in the ‘CCompiler.cpp’ source file. This class is the primary wrapper around the various tasks that our pre-processing compiler tool must undertake. Rather than require that the application be heavily involved in the compilation process, this class provides a simple interface through which much of the common functionality required by the pre-processing tool can be executed without any further involvement by the application. This common functionality includes:

1) Loading all of the required scene information from file. This information will include the scene data that is to be passed into the various compiler processes, as well as the scene data that must be exported back out to the resulting file.

2) Setting the options for each compiler task that has been selected for execution by the application.

3) Performing any high level logic for setting up the data required as input into each compilation process, and also to interpret any pieces of resulting information that may need to be merged back into the scene data set managed by the compiler class.

4) Managing and executing any compiler process modules that have been enabled by the application.

5) Writing the compiled information back out to a new file.

We have made several references in the past to individual compiler modules. Recall that one of the development goals of this tool was to create a modular core that could be easily extended to include

additional pre-processing tasks in the future. With this design concept in mind, each of the individual compilation / processing tasks that we will be implementing in this tool are separated into individual classes that are each responsible for performing only a smaller portion of the overall scene compile process. The following list outlines some of the modular processing classes that we will encounter. The first three in this list are covered in this lesson, and the final two will be covered in the next.

CProcessHSR
This procedure in effect pre-processes our scene geometry and removes any 'Illegal Geometry' ready for processing by the BSP compiler. These fragments of illegal geometry are described as ‘hidden surfaces’ due to the fact that any surfaces – or fragments of surfaces – that intersect the solid area of another part of the scene cannot usually be seen due to depth-based pixel culling. These fragments are potentially destructive and often cause complete failure when attempting to generate solid and empty leaf information within the BSP compiler. For this reason, these illegal polygon fragments must be removed.

This process works at the mesh level, meaning the scene data must be in its primitive form on import. Once processing is complete, all illegal geometry should have been removed, and the mesh data merged into a single set of polygons ready for BSP compilation.

CBSPTree
The leaf-based BSP compile process is used, among other things, for partitioning the scene (along each polygons plane) and collecting a series of convex polygon clumps, or leaves, that make up the scene. These 'nodes' and 'leaves' are then utilised by many procedures we may wish to perform on our scene database, such as portal and PVS compilation, constructive solid geometry (including HSR) and much more. The tree itself can then also be used at runtime for many operations such as collision detection and line of sight.

CProcessTJR
T-Junctions are often (not exclusively) the result of a surface being split, while its neighbour remains intact. This has the effect of splitting the edge of one surface, adding at least one vertex, while the edge of the adjacent polygon remains in tact. The problem with this situation is that it often results in artifacts during rasterization due to the differing amounts of floating point errors that occur while traversing each of these edges. These artifacts are often called 'Sparklies' or 'Sub-Pixel Gaps' where certain pixels along those edges are not rendered. This allows whatever color pixels may have been rendered behind to show through. This is not the only problem caused by T-Junctions however. Because the vertices of these two surfaces no longer link exactly, moving the centre vertex along one edge will not adjust the adjacent edge in the same manner. This process locates such problematic edges, and attempts to repair the problem by inserting a vertex on the adjacent edge.

CProcessPRT
In order to calculate the visibility between our scene's leaves, we need to determine the shape and location of the gaps that exist between them. Once we have this information, we can then proceed to use it in order to determine what is visible through the gaps these portal polygons now occupy.

Essentially this procedure feeds extremely large initial surfaces through the tree at each node, clipping it to every other subsequent node, as it traverses the tree. This process continues until the fragments of such surfaces reach an empty leaf. These remaining fragments, assuming they pass final validation,

accurately describe these gaps and become portals between our scene's leaves ready for visibility processing among other things. **Note: This process is not discussed until the next lesson.**

CProcessPVS
The PVS compilation procedure ultimately determines the 'potential' visibility between the leaves within our scene. For every leaf contained within the BSP tree, this process will determine which other leaves in the scene are visible from any position within that particular leaf. This is an extremely intensive process and can take several orders of magnitude longer to complete than every other compile process combined. **Note: This process is not discussed until the next lesson.**

The CCompiler Class

In this lab project we will only need one instance of a 'CCompiler' object that will be created and managed by the application front end discussed in the previous section. Although this application will only ever use one instance of this type, the compiler class has been designed to provide an independent, standalone interface that allows the application to easily manage multiple compiler class instances. This would allow the application to use multi-threading in order to maintain more than one compiler process that can be run concurrently – compiling multiple levels at any one time – or to simply allow the compiler to run in the background whilst the user carries on working with the tool / application into which the compiler core has been integrated.

Before we move on to the implementation of the compiler class functionality, let us begin by examining the declaration for the main compiler class. We have removed the list of member functions to improve readability, so check the source code for a full list of methods. All we are interested in at the moment are the member variables and structures declared within the class scope.

```
class CCompiler
{
public:

    // Constructors & Destructors for This Class.
             CCompiler();
    virtual ~CCompiler();

    // Public Variables for This Class.
    vectorMesh      m_vpMeshList;       // A list of all meshes loaded.
    vectorTexture   m_vpTextureList;    // A list of all textures loaded.
    vectorMaterial  m_vpMaterialList;   // A list of all materials loaded.
    vectorEntity    m_vpEntityList;     // A list of all entities loaded.
    vectorShader    m_vpShaderList;     // A list of all shaders loaded.

    // Public Member Functions Omitted

private:

    // Private Variables for This Class.
    HSROPTIONS      m_OptionsHSR;       // Hidden Surface Removal Options
    BSPOPTIONS      m_OptionsBSP;       // BSP Compilation Options
    TJROPTIONS      m_OptionsTJR;       // T-Junction Repair Options
```

```
    ILogger         *m_pLogger;          // Interface used to log progress etc.
    LPTSTR           m_strFileName;      // The file used for compilation
    COMPILESTATUS    m_Status;           // The current status of the compile run
    ULONG            m_CurrentLog;       // Current logging channel for messages.

    CBSPTree        *m_pBSPTree;         // Our compiled BSP Tree.

    // Private Member Functions Omitted

};
```

There are several important member variables declared within this class that we must first examine, so let us briefly discuss each of them.

vectorMesh m_vpMeshList
This member variable will be used to store a list of all of the meshes loaded from the source IWF scene file. It is of type 'vectorMesh', which is a typedef for an STL vector of 'CMesh' pointers.

```
typedef std::vector<CMesh*>       vectorMesh;
```

This STL vector is populated by the IWF loading code covered later in this lesson. Each mesh object stored within this array will potentially be processed by the compiler as each compilation module is executed.

vectorTexture m_vpTextureList
vectorMaterial m_vpMaterialList
vectorEntity m_vpEntityList
vectorShader m_vpShaderList
These four member variables serve as the means by which our compiler stores the static scene information that will not be processed by the compiler portion of the application. Instead this information is only maintained in order for it to be exported back out to the destination file once the scene has been processed. The type of each of these member variables is a typedef of one of the STL vector items outlined in the following list.

```
typedef std::vector<iwfTexture*>  vectorTexture;
typedef std::vector<iwfMaterial*> vectorMaterial;
typedef std::vector<iwfEntity*>   vectorEntity;
typedef std::vector<iwfShader*>   vectorShader;
```

Due to the fact that we are simply maintaining this information so that the final scene can be exported correctly, each of these STL vector items is simply designed to store pointers to each type of storage class provided directly by the IWF import library. As with the mesh information, these STL vectors are populated by the IWF loading code covered later in this lesson.

HSROPTIONS m_OptionsHSR
Each compiler module in this application has its own unique options structure that can be used by the application to alter the way in which each module performs its task – if it is to be executed at all. These structures can be found in the 'CompilerTypes.h' header file in the 'Compiler Source' project directory.

Although we do not alter these option values from their default state in this lab project, an application can access and modify the information stored in these options structure members by calling the appropriate 'GetOptions' or 'SetOptions' functions defined by the 'CCompiler' class.

This member variable contains those options available to the application for the hidden surface removal process implemented within this lab project.

```
typedef struct _HSROPTIONS
{
    bool Enabled;

} HSROPTIONS;
```

We can see that the 'HSROPTIONS' structure contains only a single Boolean variable named 'Enabled'. This variable can be used by the application to specify whether or not the scene data should be passed through the hidden surface removal process. By placing a value of 'true' into this variable we are instructing the compiler that we would like to enable the HSR process in order to ensure that all of the meshes are unioned together into a single valid mesh ready for BSP compilation.

BSPOPTIONS m_OptionsBSP

In a similar fashion to the hidden surface removal module, there are also a set of options available that allow us to alter how the binary space partitioning module will compile the final BSP leaf tree. This member variable stores these BSP module options and is declared with the type 'BSPOPTIONS'. This structure is outlined below.

```
typedef struct _BSPOPTIONS
{
    bool            Enabled;
    unsigned long   TreeType;
    float           SplitHeuristic;
    unsigned long   SplitterSample;
    bool            RemoveBackLeaves;
} BSPOPTIONS;
```

In addition to the 'Enabled' flag – used to instruct the compiler as to whether or not this process should be performed – the 'BSPOPTIONS' structure defines several options that dictate how the final scene BSP tree will be compiled. The purpose of each of these option values are detailed in the following list.

- *TreeType*
  In the previous spatial hierarchy compilation classes that we have implemented, we exposed the ability to compile a tree that did not split the polygon data against the separating planes as the tree was being constructed. While compiling the tree without splitting polygon data was not suitable for the hardware rendering techniques we constructed to handle the tree data, there were many non-rendering situations in which storing unsplit polygon data in the leaves of the tree may have been beneficial to us.

  If you have read the textbook for this chapter, you should know that splitting the polygon data during the compilation of a polygon aligned BSP leaf tree is critical and cannot be avoided. This

is due to the fact that the splitting of the source polygon data is a key aspect of how the BSP tree forms its convex leaf areas in addition to providing valid solid and empty leaf information. Although the splitting of polygons cannot be avoided during the compilation of the BSP tree itself, it is possible to adapt the way that polygon data is stored in the leaves in order to achieve a similar effect. We will not go into detail about how this process works at this point and we will defer this discussion until the coverage of our new BSP tree compilation class. Just know for now that the BSP tree compiler will have the ability to generate a tree which can store unsplit polygon information.

This option variable is intended therefore to instruct the BSP compiler as to whether or not it should generate the final scene tree using this non-split concept. This tree type variable should be set to one of the two defined constants shown below.

```
#define BSP_TYPE_NONSPLIT   0
#define BSP_TYPE_SPLIT      1
```

The first of these two constants instructs the compiler to generate a tree that stores unsplit polygon data, and the second will cause the BSP tree to be generated using the more traditional split polygon approach.

Note: If the BSP tree is intended for use by a rendering engine, then it is recommended that you select the 'BSP_TYPE_SPLIT' option. Because the non-split solution will result in certain polygons ending up in multiple leaves, selecting the traditional splitting option will prevent duplicate polygons from being rendered when using a static vertex / index buffer solution.

- *SplitHeuristic*
  This variable defines the splitting heuristic value that will be passed into the BSP compiler's 'SelectBestSplitter' function. We have already discussed the purpose of this value in detail earlier in this lesson, but to summarise this value allows us to specify the importance of building a balanced BSP tree versus reducing the number of polygons which get split during the compilation process. A good default value to specify for this heuristic variable is one of around 3.0 although you should experiment with this in order to get the best results for the specific requirements of your application.

- *SplitterSample*
  In addition to the SplitHeuristic parameter, the 'SelectBestSplitter' function also accepts a SplitterSample value which can be specified by the application through this option structure member. Recall that this concept allows us to specify the maximum number of potential polygons that will be considered for use as a separating plane by this function. This is simply an early out mechanism that allows us to select a polygon from the first 'n' polygons in the list passed to the selection function rather than testing the entire set. By reducing the number of potential splitters tested during this select step we can greatly reduce the time it takes to compile the polygon aligned BSP tree. This can be useful during the development of our application in which we might only want to build the tree quickly for testing purposes. This member can be set to 0 however when we are compiling a *release* version of the level, in order to have the 'SelectBestSplitter' function test every polygon in the specified polygon list.

- *RemoveBackLeaves*
  When compiling a polygon aligned BSP leaf tree, we have the added advantage of being able to determine the solid and empty areas within the compiled scene. As you should know if you have already been through the textbook for this chapter, the integrity of this solid information is entirely dependant on the validity of the geometry being used. If the scene geometry has been created in a manner compatible with the solid BSP leaf tree, then we should find than no polygons end up behind any terminal node within the tree structure. There are situations in which this is not the case however, even when the input geometry is valid. Due to the inherent inaccuracy of single precision floating point values on today's processors, the possibility still exists that there will be slight errors within either the source polygon data, or those polygons generated during the BSP compilation process. These small, potentially accumulating errors could lead to a situation in which a small fragment of a polygon ends up behind a terminal node that would traditionally be considered solid space.

  This flag allows the application to specify whether or not it would like the BSP compiler to enforce the removal of any polygon fragments which end up behind a node where solid space should exist. Whilst this technique does resolve many situations in which erroneous polygon fragments might fall into solid space, it should not be considered to be a replacement for hidden surface removal.

  Of course, if your application is not interested in maintaining this solid and empty space information, then it is possible to set this variable to a value of 'false'. In this case, a leaf will be generated behind those terminal nodes in which any polygons are found to exist. This is done in a manner similar to that when creating a leaf that will be attached to the front of a node.

  Again, we will look at how this process is implemented later in this chapter during the coverage of our new BSP tree compiler class.

As with each of these compiler option member variables, the 'm_OptionsBSP' member can be retrieved and altered by the application via the 'GetOptions' and 'SetOptions' compiler functions.

TJROPTIONS m_OptionsTJR

The final member variable used to store the options for an individual compiler process is the 'm_OptionsTJR' variable. This variable is of the type 'TJROPTIONS' which describes any settings available to the application for the T-junction repair compiler module. Similar to the 'HSROPTIONS' structure, this contains only a single Boolean variable that instructs the compiler as to whether or not the T-junction repair process should be enabled. In the interest of completeness, this structure is shown below.

```
typedef struct _TJROPTIONS
{
    bool Enabled;

} TJROPTIONS;
```

ILogger * m_pLogger

Earlier we discussed how the compiler portion of the application integrates itself with the front end interface using a base class named 'ILogger'. This interface is used throughout the various compiler modules in order to report any progress, success or failure information to the user. Recall that the 'ILogger' interface is a pure abstract base class that cannot be instantiated in its own right. This forces the application to derive a class from this base interface in which the code required to display the compiler's status information to the user is implemented. In order to maintain the independence of the compiler core from the application front end, this member variable stores a pointer to an instance of the applications log output object using the type 'ILogger' rather than a pointer to any application specific class. This variable should be set by the application with a call to the 'CCompiler::SetLogger' method, passing a pointer to its own logging class instance.

LPTSTR m_strFileName

This member variable stores a string that specifies the path and file name of the IWF file to be imported and processed. This variable is populated by the application using a call to the 'SetFile' method of the compiler class. This variable should be set *before* the compilation process is triggered as this string is read by the compiler when importing the scene geometry that will be subsequently passed through each processing module.

COMPILESTATUS m_Status

Earlier in this section we discussed how the compiler class was designed to function in a multi-threaded environment as either a single background process, or to run concurrently with other compilation tasks. In order to afford the application some control over the execution of each compilation process, this class exposes the ability to both query and modify the running state of the object in order to pause, resume or cancel the currently executing operation. We will discuss how the application can manipulate the state of the compiler shortly.

The 'm_Status' variable stores the current state of this compiler object and can be retrieved by the application with a call to the 'GetCompilerStatus' method of this class. This may contain any one of the four values declared in the following enumerator listing.

```
enum COMPILESTATUS
{
    CS_IDLE         = 0,
    CS_INPROGRESS   = 1,
    CS_PAUSED       = 2,
    CS_CANCELLED    = 3
};
```

Although the meaning of each status item should be relatively obvious given its name, let us just briefly examine these values and the situations in which they may be returned.

- *CS_IDLE*
  This enumerator item specifies that the compiler is currently in an idle state. This means that no compilation process is currently executing. A value of CS_IDLE will only be returned in those cases both before and after the 'CompileScene' function has been executed in order to run the compilation process.

- *CS_INPROGRESS*
  This item specifies that the compiler object is currently in the process of compiling the scene data. As soon as the application calls the 'CompileScene' function, the 'm_Status' variable will be assigned this value to signify that it has begin the operation. If this value is returned, it also means that the compiler is not currently in a paused state and has also not been cancelled by the user.

- *CS_PAUSED*
  As mentioned, the application is able to put the compiler object into a paused state that will temporarily halt any currently executing compilation task. This process can be resumed once again at a later stage should the application choose to do so. We will discuss the method by which this is achieved shortly.

- *CS_CANCELLED*
  In addition to pausing the compilation process, the application also has the ability to cancel the operation all together. Once this cancel request has been received, the 'm_Status' variable will be updated to contain this value. This informs both the various compilation modules, in addition to the application, that the task is currently in the process of being cancelled. As soon as the executing compiler module has been able to gracefully clean up and exit, the status variable will once again be assigned a value of 'CS_IDLE'. During the time in which the status variable contains the 'CS_CANCELLED' value, the application should wait for the compiler to finish before taking any further action with this compiler object.

ULONG m_CurrentLog
This member variable is maintained in order to store the log channel index for the compilation module that is currently executing. When the application's running state is altered, such as having been paused or cancelled, this variable is used to allow the compiler object to output a notification to the log channel for the current task.

CBSPTree * m_pBSPTree
The primary goal of our pre-processing tool is to build a polygon aligned BSP leaf tree based on the information imported from the IWF file specified by the application. Should the application have enabled the BSP compilation process, this member variable will store a pointer to the 'CBSPTree' object that has been compiled from this scene data.

Let us now discuss the functions to the CCompiler class that we have not yet seen. We will start by looking at the initialization functions – such as the constructor, and the member variable accessor functions. We will then examine the methods charged with importing and processing the level data.

CCompiler::CCompiler()

We begin our coverage of the 'CCompiler' methods with the class constructor. This is the function in which each of the compiler class member variables and structures are initialized with a sensible set of default values. In lab project 16.2, the front end interface does not require that the user specify any of the compilation options we discussed earlier. As a result, the compiler and each compilation process will use those default settings specified by this function.

```
CCompiler::CCompiler()
{
    // Set up default HSR options
    m_OptionsHSR.Enabled            = true;

    // Set up default BSP options
    m_OptionsBSP.Enabled            = true;
    m_OptionsBSP.TreeType           = BSP_TYPE_SPLIT;
    m_OptionsBSP.SplitHeuristic     = 3.0f;
    m_OptionsBSP.SplitterSample     = 60;
    m_OptionsBSP.RemoveBackLeaves   = true;

    // Set up default TJR Options
    m_OptionsTJR.Enabled            = true;

    // Reset Vars
    m_strFileName = NULL;
    m_pBSPTree    = NULL;
    m_pLogger     = NULL;
    m_Status      = CS_IDLE;

}
```

We begin by enabling both the hidden surface removal processor and BSP compilation modules. In addition to enabling the latter of these two processes, we also specify the default values for the remainder of the BSP compiler settings.

In this tool, we are intending for the exported data to be loaded by a rendering application. Bearing in mind our previous discussion of the non-split BSP tree option, we first specify that we would like to compile a standard tree in which the traditional *split* polygon fragments are stored within its leaves. Next we specify a reasonable split heuristic value of 3.0 and a maximum splitter sample count of 60 to ensure that the BSP compilation is relatively quick and efficient. Finally we enable the removal of any polygon fragments that end up in what should be solid space during the BSP compilation process, by setting the 'm_OptionsBSP.RemoveBackLeaves' variable to true.

We finish initializing the compiler process options variables by finally enabling the T-junction repair process. This module has no further options.

The final part of the constructor is responsible for setting the remaining member variables to their initial states. This includes setting any pointer variables to NULL to ensure that we do not attempt to access an invalid memory location at some future point, as well as setting our compiler's current status variable to that of an idle state.

CCompiler::SetFile

This function is called in order to inform the compiler of the path and filename to the IWF formatted scene file that will be compiled. As we observed in our coverage of the application front end, the 'SetFile' function should be called by the application prior to beginning the actual process of compiling the scene data.

In this function we begin by releasing any string data found to already exist within the 'm_strFileName' variable. This prevents any previously allocated memory from being leaked when we overwrite the pointer value stored in this member in the next step – which is to duplicate the file name string passed as the single parameter to this function. This duplication is achieved by calling the '_tcsdup' function which returns a pointer to a newly allocated copy of the specified string.

```
void CCompiler::SetFile( LPCTSTR FileName )
{
    // Release any old filename
    if ( m_strFileName ) free( m_strFileName );
    m_strFileName = NULL;

    // Duplicate the filename
    m_strFileName = _tcsdup( FileName );

}
```

> **Note:** Remember that when releasing any memory allocated through a call to this string duplication routine, we must use the 'free' function rather than using the more common 'delete' method.

Once we have a copy of this file name string, the compiler will be able to read the contents of the 'm_strFileName' member variable at a later stage in order to determine which file to import. We duplicate this character string, rather than simply storing the specified pointer, because we are unable to guarantee that the string passed to the 'FileName' parameter will not go out of scope and be released before the compiler has had a chance to import the file.

CCompiler::SetOptions / GetOptions

As we know, each compilation processing module in this application has its own unique options structure that can be specified by the application in order to alter the way in which each module performs its task. The two public 'SetOptions' and 'GetOptions' methods are the means by which the application can access and set the options structures for each process module available within the compiler. The first of these two functions – 'SetOptions' – accepts two parameters. The first of these parameters accepts a constant value that specifies the compiler process for which the calling function would like to set the options. The available constant values that can be passed into this parameter are found in the 'Common.h' header file and are shown below:

```
#define PROCESS_HSR         0   // Hidden Surface Removal
#define PROCESS_BSP         2   // Binary Space Partition
#define PROCESS_PRT         3   // Portals
#define PROCESS_PVS         4   // Potential Visibility Set
#define PROCESS_TJR         5   // T-Junction Repair
```

As you can see, there are six constant values defined in this list that match up with each of the processor modules described earlier in this section. Only three of these modules are applicable to this lab project however – 'PROCESS_HSR' for hidden surface removal, 'PROCESS_BSP' for BSP compilation and 'PROCESS_TJR' for the T-junction repair process. The remaining two process modules will be discussed and implemented in the next lesson. This demonstrates however, that each process has a

unique index value that can be used to identify a particular compiler module when interacting with the compiler interface through functions such as those outlined here.

The second parameter declared by this method is a void pointer named ‘Options’, into which the calling function should pass a pointer to the options structure it would like to set. This structure should be of a type applicable to the process specified by the ‘Process’ parameter because this information will be used to cast the void pointer to the correct structure type before assigning that cast structure to the appropriate compiler member variable.

```
void CCompiler::SetOptions( UINT Process, const LPVOID Options )
{
    switch (Process)
    {
        case PROCESS_HSR:
            m_OptionsHSR = *((HSROPTIONS*)Options);
            break;

        case PROCESS_BSP:
            m_OptionsBSP = *((BSPOPTIONS*)Options);
            break;

        case PROCESS_TJR:
            m_OptionsTJR = *((TJROPTIONS*)Options);
            break;

    } // End Switch
}
```

The ‘GetOptions’ method functions in a similar manner to that of the ‘SetOptions’ method. This function accepts the same two parameters. The only difference between these two functions is that the ‘Options’ parameter is now used to pass information back *out* to the calling function, rather than to receive it. Similar restrictions apply to the type of the structure passed to the ‘Options’ parameter. This should match the appropriate options structure type for the specific module passed to the ‘Process’ parameter as before. In the following code block you can see how the void pointer specified by the ‘Options’ parameter is cast to the appropriate type – based on the value contained in the ‘Process’ function parameter – and the data contained within the relevant class member variable is copied into that structure owned by the calling function.

```
void CCompiler::GetOptions( UINT Process, LPVOID Options ) const
{
    switch (Process)
    {
        case PROCESS_HSR:
            *((HSROPTIONS*)Options) = m_OptionsHSR;
            break;

        case PROCESS_BSP:
            *((BSPOPTIONS*)Options) = m_OptionsBSP;
            break;

        case PROCESS_TJR:
            *((TJROPTIONS*)Options) = m_OptionsTJR;
```

```
            break;

    } // End Switch
}
```

CCompiler::PauseCompiler / ResumeCompiler / StopCompiler

One of the features of the compiler core we are developing in lab project 16.2 is its additional support for use in a multi-threaded environment. During the earlier coverage of the 'm_Status' member variable declared by this class, we discussed the fact that the compiler class exposed the ability for the application to alter the state of the compiler object by pausing, resuming or cancelling the current operation. The three methods outlined here are the means by which the application can control this behavior.

The first of these is the 'PauseCompiler' method. In this function we begin by testing the current status of the compiler object. Due to the fact that the act of pausing the process implies that it is already running, we return immediately of the current status variable contains any value other than 'CS_INPROGRESS'. If this test passes, we then update the value of the 'm_Status' variable such that it now represents the paused state. The final task undertaken by this function is to notify the user of the update to the current compilation module's status. This is done by printing a 'Pausing…' message to the user with a call to the logging interface' 'LogWrite' method.

```
void CCompiler::PauseCompiler( )
{
    if ( m_Status != CS_INPROGRESS ) return;
    m_Status = CS_PAUSED;

    // Write Log Info
    if ( m_pLogger )
    {
        // Rewind and write 'pausing' message
        m_pLogger->Rewind( m_CurrentLog );
        m_pLogger->LogWrite( m_CurrentLog, LOGF_WARNING | LOGF_ITALIC, false,
                             _T("Pausing..."));

    } // End if Logger Available
}
```

One thing that should be clear about this function is that it does nothing more than essentially update the compiler's 'm_Status' member, followed by the output of a notification message to the user. At no stage however, does it actually pause any currently executing compilation process. This is due to the fact that each individual module is responsible for testing the current state of the compiler and taking any appropriate action based on that information. The majority of this is achieved through a call to one of the additional compiler functions named 'TestCompilerState' method that we shall be covering shortly.

The next status function available to the application is the 'ResumeCompiler' method. Applicable only if the compiler is currently in a paused state, this function updates the current status of the compiler object only if this is found to be the case. As with the 'PauseCompiler' function previously outlined, this function finally also prints an updated status message to the log channel of the currently executing

process. This function is not responsible for physically resuming the execution of the active compiler process. As before, this is the responsibility of each individual processing module and the 'TestCompilerState' method covered shortly.

```
void CCompiler::ResumeCompiler( )
{
    if ( m_Status != CS_PAUSED ) return;
    m_Status = CS_INPROGRESS;

    // Write Log Info
    if ( m_pLogger )
    {
        // Rewind and write 'resuming' message
        m_pLogger->Rewind( m_CurrentLog );
        m_pLogger->LogWrite( m_CurrentLog, LOGF_WARNING | LOGF_ITALIC, false,
                             _T("Please Wait..."));

    } // End if Logger Available
}
```

The final of the three status update functions is the 'StopCompiler' method. This can be called by the application in order to have the currently executing compiler process cancelled. As with each of the previous two methods, this function is only responsible for updating the 'm_Status' member variable and notifying the user of this updated status. In this case, a value of 'CS_CANCELLED' is used to update the status member only if the compiler process is not idle – e.g. is currently equal to CS_PAUSED or CS_INPROGRESS.

```
void CCompiler::StopCompiler( )
{
    if ( m_Status == CS_IDLE ) return;
    m_Status = CS_CANCELLED;

    // Write Log Info
    if ( m_pLogger )
    {
        // Rewind and write 'pausing' message
        m_pLogger->Rewind( m_CurrentLog );
        m_pLogger->LogWrite( m_CurrentLog, LOGF_WARNING | LOGF_ITALIC, false,
                             _T("Cancelling..."));

    } // End if Logger Available
}
```

With each of these three methods it is important to bear in mind that, while the compiler object's status variable is updated immediately, the currently executing compilation module may not be in a position to respond to this update until a much later point in time. In the interests of efficiency, each module implemented by this lab project will only test the state of the compiler at periodic intervals. This periodic testing can unfortunately lead to a slight delay in which the application must sometimes wait.

CCompiler::TestCompilerState

There are several calls to the 'TestCompilerState' method scattered throughout each of the compiler modules developed in both this lesson and the next. This function is called by the various compiler modules in order to respond to any compiler status change that may have occurred.

When this function is called by the compiler module, the first part of this function deals with the situation in which the compiler has been put into a paused state. As we can see in the following block of code; after informing the user that the compiler has been successfully placed into a paused state this function enters a while loop that will only exit following a call to 'ResumeCompiler' method that will return the 'm_Status' variable to a state of 'CS_INPROGRESS'. Inside this continuing loop, a call is made to the runtime 'sleep' function in order to have the compiler thread wait 500 milliseconds before testing the 'm_Status' member again. By calling the sleep function in this way, we prevent this compiler thread from consuming a great deal of the CPU resources available to our application.

```
bool CCompiler::TestCompilerState( )
{
    if ( m_Status == CS_PAUSED )
    {
        // Write Log Info
        if ( m_pLogger )
        {
            // Rewind and write 'pausing' message
            m_pLogger->Rewind( m_CurrentLog );
            m_pLogger->LogWrite( m_CurrentLog, LOGF_WARNING | LOGF_ITALIC, false,
                                 _T("Paused..."));

        } // End if Logger Available

        // Sleep the thread until compiler is resumed
        while (m_Status == CS_PAUSED) Sleep(500);

    } // End if Paused
```

Assuming the application had not been placed into a paused state, it is possible that the compile process had been cancelled. If this is found to be the case, a value of 'false' is returned to the compiler module making the call to this function in order to inform it of the fact that it should clean up and exit gracefully. In all other cases this function simply returns a value of true to denote that the compile process should continue to execute.

```
    // Should we cancel ?
    if ( m_Status == CS_CANCELLED ) return false;

    // Continue running.
    return true;
}
```

CCompiler::CompileScene

The 'CompileScene' function is the means by which the application can instruct the compiler to execute each of the various compilation modules that may have been enabled. This function should be called by the application only after having selected and informed the compiler of the file that is to be imported and

processed. If any customization of the various module options is required, these should also have been set prior to this function being called. This function accepts no parameters, and returns a single Boolean result that signifies whether the compilation process as a whole was successful or not.

Although not strictly necessary for this method to function correctly, the first task undertaken here is to retrieve the filename portion of the path string stored in the 'm_strFileName' member variable. With the path stripped from the import file string; this will be used only as part of the information output to the compile process log. This is achieved by searching for the *last* forward or backslash contained in the string and – if one exists – retrieving a pointer to the portion of the string which immediately follows that character.

> **Note:** In order to prevent any obfuscation of the logic contained within this and other class methods, much of the logging code has been omitted and replaced with a single line comment outlining the information that would be output to the log at that point. Refer to the source code in the lab project archive provided with this lesson if you wish to view the source files with all log output code included.

```
bool CCompiler::CompileScene( )
{
    CFileIWF IWFFile;

    // Validate Data
    if (!m_strFileName) return false;

    // Retrieve the filename portion only of the string
    LPTSTR FileName = NULL;
    FileName = _tcsrchr( m_strFileName, _T('\\') );
    if (!FileName) FileName = _tcsrchr( m_strFileName, _T('/') );
    if (!FileName) FileName = m_strFileName;
    if (FileName[0] == _T('\\') || FileName[0] == _T('/')) FileName++;
```

The first real job that the 'CompileScene' function must perform is to import the scene information from the IWF file specified by the application. In previous lab projects we have used the 'CFileIWF' class provided by the IWF import library to perform the file handling and data loading steps. Our pre-processing tool is no different in this regard. At the start of this function we have defined an object named 'IWFFile' which is of the type 'CFileIWF'. Although we are using a custom IWF loading class in this application rather than that provided by the libIWF library, the concepts are the same in both cases. As we have done many times in the past, we instruct the IWF loading object to import the data from the IWF file by calling the 'CFileIWF::Load' method, passing in the name and path of the scene data file to be imported as the single parameter input to this function.

```
    try
    {
        // We Are starting the process
        m_Status = CS_INPROGRESS;

        // Load the specified file
        IWFFile.Load( m_strFileName );
```

As has been the case with the 'CFileIWF' class used in each of our previous lab project applications, this class maintains a list of the individual pieces of information loaded from the file within internal STL

vector members. In our rendering applications, it has traditionally been the responsibility of the 'CScene' class to then iterate through these arrays and process each piece of information in an effort to build renderable data where applicable. However, our pre-processing tool is not required to perform any kind of rendering; we simply need to store this information such that we can gain easy access to the file information at any point in the future.

Because of the fact that the STL vector variables within the 'CFileIWF' class are defined with an identical type to those corresponding members contained within our compiler class, making a copy of this data is made extremely simple. As a result of this typed similarity, we can simply take ownership of the information stored in the 'CFileIWF' class instance by assigning the matching member variables in our compiler class to those STL vectors stored in the file object as demonstrated in the following code.

```
        // Obtain ownership of the objects loaded
        m_vpMeshList     = IWFFile.m_vpMeshList;
        m_vpMaterialList = IWFFile.m_vpMaterialList;
        m_vpTextureList  = IWFFile.m_vpTextureList;
        m_vpShaderList   = IWFFile.m_vpShaderList;
        m_vpEntityList   = IWFFile.m_vpEntityList;
```

At this stage we have made a copy of the data stored within the IWF file object into our compiler class's internal member arrays. However, recall that these STL vectors simply store pointers to heap allocated storage classes. This means that by making a simple shallow copy of the array from one class instance to the other, both sets of arrays will now contain elements that reference the same instance of each data object. If the IWFFile object was to go out of scope and be released at some point in the future, this would have the unfortunate side effect of releasing all of the objects now referenced by the compiler class too. To prevent this, we must clear the IWF object's STL vector members to ensure that the compiler object remains the only place in which these data items are referenced.

```
        // Clear out the IWF's object vectors (don't destroy)
        IWFFile.m_vpMeshList.clear();
        IWFFile.m_vpMaterialList.clear();
        IWFFile.m_vpTextureList.clear();
        IWFFile.m_vpShaderList.clear();
        IWFFile.m_vpEntityList.clear();

        // Log - General: 'FileName' was imported succesfully

    } // End try Block
```

Notice in the previous code how the file loading logic was wrapped in a try block that will attempt to catch any exceptions which may have been thrown by the 'CFileIWF::Load' function, or any of the subsequent STL vector operations. The following catch blocks are provided to ensure that all memory allocated to date by both the IWF object during the loading procedure, in addition to any memory allocated by the compiler class are fully released before we return a failure code from this function.

```
    catch (HRESULT & e)
    {
        // Log - General: Import of 'FileName' failed with error 'e'
```

```
        IWFFile.ClearObjects();
        Release();
        return false;

    } // End Catch Block
    catch (...)
    {
        // Log - General: Import of 'FileName' failed

        IWFFile.ClearObjects();
        Release();
        return false;

    } // End Catch Block
```

At this point in the execution of the 'CompileScene' function, we have imported all of the applicable scene data from the source IWF file and have stored that information within the internal STL vector members ready for processing.

We briefly touched upon the fact that the compiler class is responsible for preparing the input data for each process earlier in this section. Rather than have the 'CompileScene' function perform each of these process specific tasks, the code for the actual preparation and execution of each processing module has been placed in a series of functions named 'PerformHSR', 'PerformBSP' and 'PerformTJR'. We will examine these functions in detail shortly, but for now we need only be aware that when any one of these three functions is called, that particular compiler process will be executed from start to finish before returning control back to this 'CompileScene' function.

In the following code we can see that this is the point at which we begin to actually process the scene data, calling each of these three processing functions in order if they have been enabled by the application. In later lessons we will be adding additional compiler modules to the pre-processing tool. For each module that we add, a new separate 'Perform*' method will need to be developed and called from within this function in a similar fashion to those shown below. Once each of the enabled processing functions has been called, we simply exit from this function and return control back to the application having completed the entire compilation process.

```
    // Log - General: Beginning compile run

    // Start compiling by removing all hidden surfaces
    m_CurrentLog = LOG_HSR;
    if ( m_OptionsHSR.Enabled && m_Status != CS_CANCELLED) PerformHSR();

    // Build the BSP Tree if requested
    m_CurrentLog = LOG_BSP;
    if ( m_OptionsBSP.Enabled && m_Status != CS_CANCELLED) PerformBSP();

    // Repair any T-Juncs if requested
    m_CurrentLog = LOG_TJR;
    if ( m_OptionsTJR.Enabled && m_Status != CS_CANCELLED) PerformTJR();

    // Clean up if required
    if ( m_Status == CS_CANCELLED ) Release();
```

```
    // Processing Run Completed
    m_Status = CS_IDLE;

    // Log – General: Compilation Success

    // Success!
    return true;
}
```

As you can see, this relatively simple function is essentially just a wrapper that first loads the scene information, and then simply triggers each of the requested processing modules. Let us therefore begin to take a look at how each of these processing modules is implemented within lab project 16.2.

CCompiler::PerformHSR

The first of the three main module functions implemented within this initial pre-processing application is the 'PerformHSR' function. This function is intended to prepare the data destined to be merged together by the hidden surface removal processor. Recall that the hidden surface removal process accepts a series of meshes that will be combined using the constructive solid geometry *union* operation. This is done in order to remove any surfaces that would introduce illegal geometry into the tree during BSP compilation. If we do not remove these illegal polygon fragments before the tree is compiled, then we end up in a situation in which the solid and empty leaf information is compromised and cannot be resolved. For this reason, the hidden surface removal process must be executed prior to compiling the final scene BSP tree. We will cover the actual hidden surface removal processing module a little later, but first we must understand the type of data that is to be passed into this process and how it is accessed.

In summary, the general process that the 'PerformHSR' function must follow is listed below:

1) Set up the hidden surface removal processor module ready for its execution.

2) Collect a list of all of the meshes in the scene that are not flagged as detail objects and pass each of them into the HSR module class ready for processing.

3) Trigger the HSR module and allow it to run its course.

4) If the HSR process was successful, each of the original non-detail meshes will now have been merged into one single resulting mesh with all hidden surfaces removed. This function must therefore release each of these original meshes from the compiler's scene database in order to ensure that the scene polygon data is not duplicated.

5) Finally, we must retrieve the single resulting mesh from the processing object that constitutes the union of each of those meshes used in the merging operation. This mesh will then be stored in the compiler's internal mesh array ready for compilation by the following BSP compilation process should it have been enabled.

It should be clear from this general outline that this function is not physically responsible for performing any part of the hidden surface removal process. Under the conditions set forth by our modular design,

each processing module is a stand alone class that has its own API. This interface is used by the compiler class in order to initialize, execute and retrieve any data that may have resulted from that module class's execution. While each processing class may differ in the way data is passed in and retrieved, this separation of functionality into individual processing classes should allow us to more easily integrate additional processing tasks into the tool we are building.

We made mention in the previous summary outline that a mesh could be flagged as a 'detail object'. Although we have not been introduced to this type of object in the past you can think of them as being somewhat like those objects that have been linked to the previous spatial tree types using the detail area concept. These are objects that are not compiled directly into the tree itself, but are instead linked at runtime by passing them through the tree and attaching them to any leaves into which they may fall. In this pre-processing tool we treat these objects in a similar fashion. Although the objects will eventually be linked to the leaves only in our rendering application, we must ensure that any scene meshes that have been flagged as detail objects are not altered or compiled into the resulting BSP leaf tree.

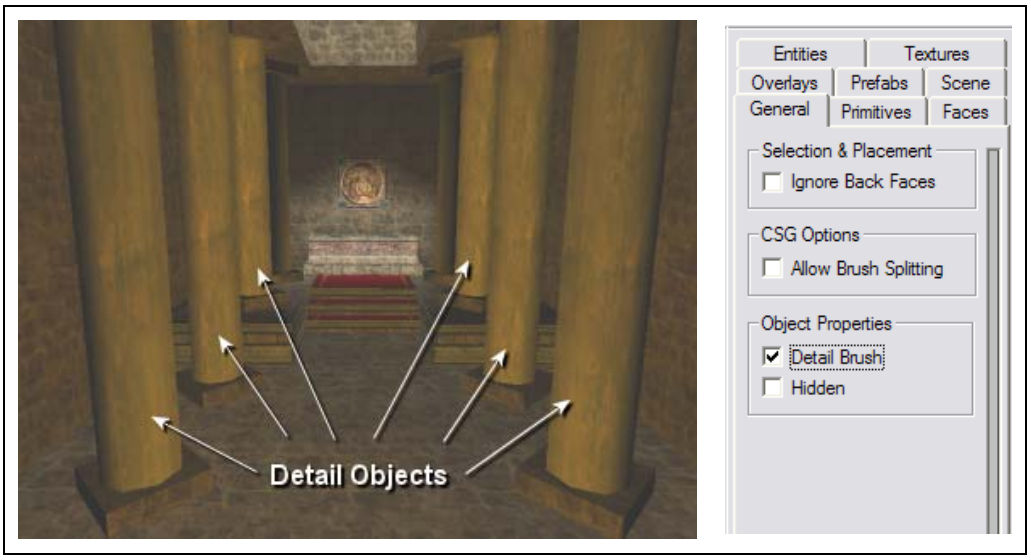

Figure 16.4

Figure 16.4 demonstrates the type of situation in which we might decide to make use of the detail object concept when compiling a level using the polygon aligned BSP leaf tree compiler. This scene contains a series of high resolution cylinders used to depict several roof support columns. Objects – such as these cylinder meshes – which contain a high level of detail, or a large amount of variation in the orientation of their polygons, can be ideal candidates for selection as detail objects. This is due to the fact that objects of this type can potentially introduce many hundreds of nodes and leaves when included in the BSP tree compilation process, without providing any significant benefit to the tree structure. In this figure we can also clearly see that these column meshes would not provide a significant amount of occlusion information for the visibility module (developed in the next lesson) that might have made their inclusion in the tree worthwhile.

On the right hand side of this figure we have included an image of the general properties tab from the GILES™ level editing tool. We can use this editor to select any mesh in our IWF scene, and flag it to become a detail object by checking the 'Detail Brush' box highlighted. After saving scene back out to

file, when we subsequently load it back into our rendering application or compiler tool we can check for the existence of this flag using the following bit test:

```
if ( pMesh->Flags & MESH_DETAIL ) { // This is a detail object }
```

Another good example of common types of detail objects are those meshes that are used as set decoration within the scene. This might include items such as tables, chairs, beds or basically anything that is not part of the scene architecture.

So, armed with this additional knowledge of what types of meshes should and should not be included in the hidden surface removal and BSP compilation processes, let us now take a look at how the 'ProcessHSR' function has been implemented in this lab project.

```
bool CCompiler::PerformHSR()
{
    // One time compile process
    CProcessHSR ProcessHSR;
    ULONG       i, MeshCount = 0;

    // Set our processor options
    ProcessHSR.SetOptions( m_OptionsHSR );
    ProcessHSR.SetLogger( m_pLogger );
    ProcessHSR.SetParent( this );

    // Log – HSR: Beginning Process
```

As we should have gathered from our examination of 'CompileScene', this function accepts no parameters. All of the data required by this function should have been imported into the compiler class' internal member arrays at this point, and can be access directly within this function. At the top of this function we can observe that an instance of our first processing class 'CProcessHSR' is created on the stack, as a local variable named 'ProcessHSR'. This object exposes all of the functionality required for initializing, executing and retrieving the mesh data resulting from the hidden surface removal process.

Once this processing object has been instantiated, we can then begin to initialize the various settings and values required by this module. The first of these is the 'HSROPTIONS' structure that we encountered earlier. Like the compiler class itself, each of the processing module classes – such as 'CProcessHSR' – implements a 'SetOptions' function used to set that object's option values. Recall that the 'HSROPTIONS' structure contains only a single variable that denotes whether or not this process is enabled. Regardless of this fact, we pass the options structure in at this point in order to ensure that the hidden surface removal class will have access to any additional process settings that may be added to the options structure in the future.

During each of the processing tasks, specific progress and status information must be reported to the user as we know. The next item that we pass to the HSR processing object therefore, is a reference to the logging class specified by the application. This gives the module a chance to report any process specific failures that may have occurred, in addition to providing any general information such as an operation progress percentage.

The final item that we pass into the 'ProcessHSR' object in the previous block of code is a reference to the compiler object itself. Having access to the compiler object will allow the process module to call the aforementioned 'TestCompilerState' method of the compiler as required.

With the processing module having been initialized with all of the various options and house-keeping items we can now begin to pass in the mesh data that will be used. Before we do this however, we must perform a final validation step. Of course, in order to perform a union CSG operation such as that required by the hidden surface removal process, we need at least *two* meshes in order for this operation to function correctly. If there is only one mesh in the scene to begin with, then there would be nothing else available for us to merge that mesh with. With this in mind, we first total the number of non detail meshes that have been loaded. If we find ourselves in a situation whereby there is only one available scene mesh to be processed and compiled, we simply output this information to the user and take no additional action.

```
    // Count the number of scene meshes
    for ( i = 0; i < m_vpMeshList.size(); i++ )
    {
        CMesh * pMesh = m_vpMeshList[i];
        if ( !pMesh ) continue;

        // Tally Mesh ?
        if ( !(pMesh->Flags & MESH_DETAIL) ) MeshCount++;

    } // Next Mesh

    // How many meshes ?
    if ( MeshCount <= 1 )
    {
        // Log – HSR: Nothing to do.

    } // End if single mesh
    else
    {
```

At this stage we know that we have at least two non detail meshes that must be unioned in preparation for the BSP compilation step. In the next section of code from this function we then begin to loop through each of the meshes stored in the compilers internal mesh array. If there is no mesh stored in any particular element (e.g. the mesh pointer stored there is equal to NULL) then we automatically continue to the next element in the array. It is imperative that we perform this simple test due to the fact that whenever a mesh is to be removed from consideration it is simply released before having its associated element in the mesh array set to NULL. This is done in the interests of efficiency and prevents us from having to reshuffle the mesh array at each step. We will see this behaviour shortly. In addition to testing the mesh pointer, the content of the mesh's 'Flag' member must also be tested within this loop. If this mesh is considered to be a detail object, then it should not be merged by the HSR process and so we simply move on to the next element in the array without action. With both of these tests completed, we reach a point in which we have a valid pointer to a non-detail mesh. This mesh is then passed into the HSR processing object with a call to the 'AddMesh' method of that class.

Once this loop has completed, the HSR processing module will now have a list of every mesh that we would like to merge together into one single resulting mesh, with all illegal geometry having been

removed. This is all the information this processing module needs to be aware of and so we call the object's 'Process' method to instruct the module to perform the actual hidden surface removal step on those meshes specified.

```
        // Add all of our meshes to the HSR Processor
        for ( i = 0; i < m_vpMeshList.size(); i++ )
        {
            CMesh *pMesh = m_vpMeshList[i];
            if (!pMesh) continue;
            if ( pMesh->Flags & MESH_DETAIL ) continue;
            ProcessHSR.AddMesh( pMesh );

        } // Next Mesh

        // Begin the HSR Process
        ProcessHSR.Process();
```

When the module's 'Process' function returns, it will have constructed a new mesh object that contains each of the polygons having been merged during the hidden surface removal process. This is of course unless an error had occurred. While the HSR processing class is responsible for reporting the specifics of any errors to the user via the logging class specified, we must still ensure that we take the appropriate action. To determine whether or not an error occurred during the hidden surface removal process, we test the value returned by the modules 'GetResultMesh' function. If this returns a value not equal to NULL then we know that no error occurred and we can continue to process the resulting data.

Before we actually do anything with the mesh generated by the 'Process' method, we must first remove each of the original non-detail objects that were unioned together in this step. As mentioned earlier, we do this in order to ensure that the final scene data set does not contain duplicate information. This is achieved using a method similar to that of the previous loop in which we added meshes to the processor prior to its execution. In this case however, whenever we find a valid pointer to a non detail mesh in the compiler's mesh array, we release that mesh object. Recall that we validated each element in the mesh array earlier to ensure that it did not contain a NULL pointer. The following loop logic demonstrates the reason why this must be done – once the original mesh object has been released, we subsequently set the corresponding array element to NULL. These empty elements will be removed at a later point in the application but at this stage this is done simply to prevent the mesh array from being reshuffled and resized at each step.

```
        // Make way for our result if there is one
        if ( ProcessHSR.GetResultMesh() != NULL )
        {
            // Destroy any loaded meshes
            for ( i = 0; i < m_vpMeshList.size(); i++ )
            {
                CMesh * pMesh = m_vpMeshList[i];
                if ( !pMesh ) continue;
                if ( pMesh->Flags & MESH_DETAIL ) continue;
                delete pMesh;
                m_vpMeshList[i] = NULL;

            } // Next Mesh
```

With each of the original source meshes removed, we can now retrieve the resulting mesh from the 'ProcessHSR' object and store it in the compiler's mesh array ready for the next compilation process. Rather than simply add this mesh to the end of the mesh array however, we have the ability to reuse one of the array elements that we had set to NULL in the previous loop. The following code searches for an array element containing a NULL pointer. If one is found, the index to that element is stored in the local 'MeshIndex' variable before breaking out of the loop.

```
            // Store our single result mesh in the first empty slot
            long MeshIndex = -1;
            for ( i = 0; i < m_vpMeshList.size(); i++ )
            {
                if ( !m_vpMeshList[i] ) { MeshIndex = i; break; }

            } // Next Mesh Slot
```

Notice that the local variable 'MeshIndex' was initialized using a default value of -1 in the previous code. If there were no elements found to contain a NULL pointer, this variable will not have been overwritten with a new index value and should therefore still be in its default state of -1. If an empty slot in the mesh array *was* found, this variable would now contain an index in the range of 0 and above. The following code examines the contents of this variable to determine whether it can reuse one of the available empty array elements, or if it must add the new resulting mesh to the end of the compiler's internal mesh array. If the 'MeshIndex' array contains a value of -1 then no slot was found and a pointer to the new mesh – accessed with a call to the module's 'GetResultMesh' method as before – is added to the end of the mesh array with a call to the STL vector 'push_back' function. If on the other hand the value of 'MeshIndex' is anything other than -1, then we simply overwrite that element in the array with the new mesh pointer.

```
            if ( MeshIndex == -1 )
            {
                m_vpMeshList.push_back( ProcessHSR.GetResultMesh() );

            } // End if No Slot
            else
            {
                m_vpMeshList[ MeshIndex ] = ProcessHSR.GetResultMesh();

            } // End if Free Slot
```

At this stage we have now removed every non-detail mesh originally contained in the compiler's mesh array, merged them together, and finally replaced them with a single new mesh result. With all of the illegal geometry hopefully now removed from the scene data set, we are now ready to have this new mesh processed by the next module in the list – the BSP compiler. As a result, the last part of this function simply informs the user that the HSR process has completed, and then returns to the calling 'CompileScene' function.

```
        } // End if

    } // End if multiple meshes
```

```
    // Log – HSR: Write Success Information

    // Success!!
    return true;
}
```

Although we have not yet covered the code to the hidden surface removal processor at this point, the discussion of this function will serve us well in understanding how this module fits into the larger compilation scheme. In addition to this we should now have a clearer picture of exactly what data is passed to the module class and what pieces of information the compiler is expecting to receive.

CCompiler::PerformBSP

Referring back to the code in the 'CompileScene' function, we know that this 'PerformBSP' function is called directly following the hidden surface removal step. With the HSR process completed, we are now able to compile a solid BSP leaf tree without worrying about the impact that any illegal geometry might have on the validity of the tree structure generated.

This is the stage in which we begin constructing the data that will serve as the foundation for much of the additional information generated by the future modules we will be implementing in both this lesson and the next. As we know, the 'CCompiler' class contains a member variable named 'm_pBSPTree'. This variable will store the pointer to an instance of the 'CBSPTree' class we will be discussing shortly. In the next lesson we will be using much of the information constructed in this step to generate the portal data that describes how the leaves in the tree are connected together. These portal polygons will then allow us to build the inter-leaf visibility information (PVS) which will provide us with scene occlusion culling. All of this information will eventually be stored within the BSP tree we construct in this step which will also be used as the source from which the final scene file is built.

From a high level standpoint, the 'PerformBSP' function undertakes the following tasks:

1) Set up the BSP compiler module ready for its execution.

2) Collect a combined list of all of the polygons from every non detail mesh in the scene and pass that list to the BSP tree compiler. These polygons will be used as the basis for the compilation of the polygon-aligned BSP leaf tree itself.

3) Trigger the BSP compilation process and allow it to run its course.

4) If the BSP compile process was successful, any original non-detail meshes that were used in the construction of the BSP tree should be removed. The polygon data is now owned by the centralized scene BSP tree stored within the compiler object. When exporting the scene at the end of the compilation process, the polygon data will be built using that information contained within the BSP tree.

As we can see from the above overview, the responsibilities of this method are fewer in number than those of the 'PerformHSR' function. We should already be familiar with the types of data required by

the BSP compiler at this point, so we will get straight down to examining the implementation of the 'PerformBSP' function.

```
bool CCompiler::PerformBSP()
{
    ULONG       i;

    // Destroy any old BSP Tree
    if ( m_pBSPTree ) delete m_pBSPTree;

    // Allocate a new BSP Tree
    m_pBSPTree = new CBSPTree;

    // Set our compiler options
    m_pBSPTree->SetOptions( m_OptionsBSP );
    m_pBSPTree->SetLogger( m_pLogger );
    m_pBSPTree->SetParent( this );

    // Log – BSP: Beginning Compilation Process
```

In a similar fashion to the 'PerformHSR' method, there are no parameters that need to be specified when calling this function. As before, everything we need is already contained within the member variables of the compiler class itself. Unlike this previous method however, there is no local variable declared here which serves as the processing module. In the case of hidden surface removal, the processing module class we used was instantiated as a temporary object which simply processed and built data subsequently used as a replacement in one single step. With the BSP tree class, we are building a tree structure that must be retained both for future processing modules, in addition to the requirement that this structure to be exported to file at a later point in the application. As a result, we create an instance of the 'CBSPTree' class and store it within the 'm_pBSPTree' member variable of the compiler class.

After creating the BSP class instance, we specify the compilation options that it must use with a call to the 'CBSPTree::SetOptions' method, passing in the 'BSPOPTIONS' structure stored in the compiler's 'm_OptionsBSP' member variable. We also inform the compiler module of the applications logging class object, in addition to passing the reference to this compiler object with calls to the 'SetLogger' and 'SetParent' calls respectively.

With the main BSP tree member object instantiated and initialized, we are now ready to begin processing the loaded scene information in order to compile the tree structure. In the following code we begin by looping through each of the meshes currently stored within the compiler's mesh array, adding each mesh's polygon data to the tree ready for compilation. If the hidden surface removal process was enabled and executed in the previous step, it is possible that this array will only contain a single valid mesh pointer. With that said, it is not a requirement that the HSR process be enabled in cases where the scene contains no illegal geometry to begin with, or those in which valid solid / empty information is not required. For this reason we must still loop through each mesh rather than simply searching for the first valid mesh pointer. Also recall that we do not want to include those meshes flagged as being detail objects in the compilation of the BSP tree. To this end we skip over any meshes whose 'Flag' variable contains the 'MESH_DETAIL' bit as before. Each time we find a mesh in the array that should be included in the BSP compilation process, we add its polygon data to the BSP compiler object with a call to the 'AddFaces' method. This function accepts two parameters. The first parameter should be passed

an array of ‘CFace’ object pointers – such as those managed by each mesh – each of which will be appended to the internal list of polygons to be used. The second of these two parameters should be passed a simple integer value containing the length of the aforementioned array.

With each mesh’s polygon data passed into the BSP tree object, the process if compiling the tree can now begin. In the case of the ‘CBSPTree’ class, this is done with a call to its ‘CompileTree’ method.

```
    // Add all remaining meshes polys (automatically backup with NSR)
    for ( i = 0; i < m_vpMeshList.size(); i++ )
    {
        CMesh * pMesh = m_vpMeshList[i];
        if ( !pMesh ) continue;
        if ( pMesh->Flags & MESH_DETAIL ) continue;
        m_pBSPTree->AddFaces( pMesh->Faces, pMesh->FaceCount );

    } // Next Mesh

    // Compile the BSP Tree
    m_pBSPTree->CompileTree();
```

Once the BSP tree has been successfully compiled, each of the non-detail meshes used to construct that tree should now be removed from the compiler’s scene database. If we did not perform this step then our export file would contain at least one duplicate of every polygon in the tree – one stored in the source mesh and another in the BSP tree itself. This loop simply searches for any valid non-detail mesh, releases it and finally overwrites the corresponding element in the compiler’s mesh array with a NULL pointer.

```
    // Destroy any loaded scene meshes
    for ( i = 0; i < m_vpMeshList.size(); i++ )
    {
        CMesh * pMesh = m_vpMeshList[i];
        if ( !pMesh ) continue;
        if ( pMesh->Flags & MESH_DETAIL ) continue;
        delete pMesh;
        m_vpMeshList[i] = NULL;

    } // Next Mesh
```

In the final part of this function we unset the BSP tree object’s logging class reference by passing NULL to the ‘SetLogger’ function. This is not strictly necessary within our lab project application. However, in doing so, we ensure that any BSP tree class function that outputs progress information during the main tree compile step does not output this information again in the event that any future module makes a call to it.

```
    // Prevent future logging
    m_pBSPTree->SetLogger( NULL );

    // Log – BSP: BSP Process Completed

    // Success!!
    return true;
```

```
}
```

With the BSP compilation step completed we once again return control back to the 'CompileScene' function that made the original call to this method. The next processing task undertaken by the compiler is the T-junction repair step.

CCompiler::PerformTJR

The 'PerformTJR' function is the primary wrapper around the preparation and execution of the T-junction repair process. This process constitutes the final pre-processing task that we will develop in this lesson.

The process of T-junction repair has been covered in previous lessons, so we should already be familiar with the general concepts involved. We know from experience that – at a high level – this process compares the vertices of every polygon in the scene against the edges of every other polygon that surrounds it. If no vertex is found to have a matching position within the edge being tested then a copy of that vertex must be inserted in the neighboring edge in order to resolve this T-junction situation. With this in mind, it should be clear that we must somehow pass the scene polygon information into the T-Junction repair module in order for it to have access to the level geometry. At this point in the process however, there are two locations in which the scene polygon information may exist. The first is from the mesh data contained within the compiler's internal mesh array. However, this array will only contain applicable mesh data if the BSP compilation step was bypassed. Recall that during the BSP compile step, the 'PerformBSP' function may remove those non-detail meshes that were used to construct the tree if the BSP process has been enabled by the application. As a result, the second case that we must consider is that in which the BSP tree has already been compiled and we must source the polygon data from the compiler's BSP tree object as a result.

As we know, the T-Junction repair process is not concerned with any of the renderable polygon information such as textures or materials, or even the surface normal. All this process needs to be able to function is a list of the vertices that make up each of the polygons in the scene. If we refer back to the discussion of the base 'CPolygon' class – from which each of the extended polygon data classes are derived – it should be clear that this process need only access the information exposed by that base class.

With this situation in mind, the T-Junction repair processing module has been designed to accept a list of *pointers* to one or more 'CPolygon' objects. In doing so, we allow the T-Junction repair process to gain access to the polygons from a wide variety of sources even if they are of different physical types. As long as each of the source polygon objects to be repaired are *derived* from the base 'CPolygon' class, then we can pass this data into the repair module and have the process function regardless of their type.

Before we move onto examining the internal workings of this method, let us quickly summarize the procedure as it will be implemented.

1) Set up the T-Junction repair module ready for its execution.

2) Collect a combined list of all of the polygons from every non detail mesh in the scene, or alternatively from any compiled BSP tree object, and pass that list to the T-Junction repair module ready for processing.

3) Trigger the repair process and allow it to run its course.

As you can see, this function is a relatively simple one. Let us therefore move on to briefly discuss how it has been implemented in this lab project.

```
bool CCompiler::PerformTJR()
{
    ULONG i, k;

    // One time compile process
    CProcessTJR ProcessTJR;

    // Set our process options
    ProcessTJR.SetOptions( m_OptionsTJR );
    ProcessTJR.SetLogger( m_pLogger );
    ProcessTJR.SetParent( this );

    ULONG       PolyCount = 0, Counter = 0;
    CPolygon ** ppPolys   = NULL;

    // Log – TJR: Beginning Process
```

As with each of the tasks before it, the first thing we must do in this function is to instantiate the T-junction repair module object and initialize the various options, logging and parent properties. Similar to the hidden surface removal process, T-junction repair is a one time procedure that simply manipulates any data that already exists in the scene. Once the process has completed, this processing module object is no longer required. As a result, the ‘CProcessTJR’ class is instantiated as a local variable named ‘ProcessTJR’. Once this object has been initialized we can begin to prepare any applicable scene information ready for the T-junction repair module to perform its task.

The first case we deal with in this function is that in which no BSP tree has yet been compiled. In this case there may be one or more non-detail meshes contained within the compiler’s internal scene mesh array. Because we cannot know in advance how many polygons are stored within these mesh objects in total, we must first iterate through each non-detail mesh and count up this total. If it turns out that there are no available polygons in the list, this function can simply return without error.

```
    if ( !m_pBSPTree )
    {
        // Count all polys in all active meshes
        for ( i = 0; i < m_vpMeshList.size(); i++ )
        {
            CMesh * pMesh = m_vpMeshList[i];
            if ( !pMesh ) continue;
            if ( pMesh->Flags & MESH_DETAIL ) continue;
            PolyCount += pMesh->FaceCount;

        } // Next Mesh
```

```
        // If no polys exist, bail (no error)
        if (!PolyCount) return true;
```

Now that we know the total number of polygons contained within each of the meshes in the compiler' scene database, we can begin to allocate and build the polygon pointer array discussed earlier. Our first job therefore is to allocate an array of 'CPolygon' base class *pointers* that will store any references to each polygon needed. This pointer returned by this allocation is stored in the local variable 'ppPolys' declared at the top of the 'PerformTJR' function. If this array was allocated successfully, this function subsequently collects a pointer to each polygon, from every valid non-detail mesh, and stores them in this newly allocated array.

```
        // Allocate our polygon pointer array
        if (!(ppPolys = new CPolygon*[PolyCount])) return false;

        // Add all mesh face pointers here
        for ( i = 0, Counter = 0; i < m_vpMeshList.size(); i++ )
        {
            CMesh * pMesh = m_vpMeshList[i];
            if ( !pMesh ) continue;
            if ( pMesh->Flags & MESH_DETAIL ) continue;

            // Loop through faces
            for ( k = 0; k < pMesh->FaceCount; k++ )
                ppPolys[ Counter++ ] = pMesh->Faces[ k ];

        } // Next Mesh
    } // End if Meshes
```

At this stage, assuming no BSP tree had been compiled prior to this function being called, the local 'ppPolys' variable will reference an array that contains a pointer to every polygon, taken from every non-detail mesh in the scene. If a BSP tree had been compiled however then, as we know, there will be applicable meshes remaining in the compiler's mesh array. As a result, we must also cater for the situation in which a BSP tree has been compiled.

In this case, we already know in advance how many polygons are stored in the already compiled BSP tree object. We can gain access to this information by calling the 'CBSPTree::GetFaceCount' method. Again, if the number of polygons is found to be 0 at this stage, then we return from the function without error. With the total number of polygons to hand, we allocate the 'CPolygon' pointer array on the heap and store its pointer in the local 'ppPolys' array in the same fashion as the previous case. If the allocation of this array was successful, we finish up this conditional block by filling the same polygon pointer array with references to each polygon contained in the BSP tree. These polygons are retrieved by calling the 'CBSPTree::GetFace' method.

```
    else
    {
        // Retrieve poly count
        PolyCount = m_pBSPTree->GetFaceCount();

        // If no polys exist, bail (no error)
```

```
        if (!PolyCount) return true;

        // Allocate our polygon pointer array
        if (!(ppPolys = new CPolygon*[PolyCount])) return false;

        // Loop through faces and add
        for ( k = 0; k < PolyCount; k++ )
            ppPolys[ k ] = m_pBSPTree->GetFace( k );

    } // End if BSP Tree
```

So, in either of the cases outlined above we now have an array that contains a list of polygons, referenced by pointers to their base 'CPolygon' class. This will of course be the case regardless of the actual type of the polygon objects being referenced. Armed with all the information we need, we then trigger the T-Junction repair module with a call to the 'ProcessTJR' object's 'Process' function, passing in both the array we have just constructed, along with the number of polygons stored therein. Due to the fact that we have passed in only *references* to the polygon data contained in either the mesh array, or the BSP tree, the T-Junction repair process will manipulate the polygon data without the need to be aware of the location from which they were retrieved. In addition, because the process processes and inserts vertices directly into the source polygon data, there is no need for us to retrieve and interpret any kind of resulting data. Once the repair process has completed, there is nothing further required of this function other than to release the temporary polygon array that we had previously allocated. With this allocated resource released, we finally report the process success to the user and return control to the calling function.

```
    // Repair any T-Junctions
    ProcessTJR.Process( ppPolys, PolyCount );

    // Release the poly pointer array
    if (ppPolys) delete []ppPolys;

    // Log – TJR: Process Completed

    // Success!!
    return true;
}
```

Once this function has returned, we will have come to the end of the entire compilation process for this lab project. In future lessons we will of course add additional processing modules to the compiler core that will add additional steps. In this version of our pre-processing tool however, the front end application would instruct the compiler to save the resulting information at this point. Therefore, let us now examine the compiler method used to save our newly compiled scene information to file.

CCompiler::SaveScene

The final task that our compiler must undertake once the scene data has been successfully processed is to save the compiled information back out to file.

There are several pieces of information that we have imported, compiled or processed that need to be written back out to the final IWF file. These include the texturing, material, shader and entity information we imported earlier in addition to any surviving mesh data that still exists within the compiler's mesh array. Of course, we must also export the BSP tree data to file too should that compiler process have been enabled. Let us briefly refer back to a statement that was made earlier in this chapter regarding the 'BuildFromBSPTree' function exposed by the new 'CMesh' class.

"Although we have not yet discussed our new BSP leaf tree class, we should be familiar enough with the general concepts involved in the construction of leaf trees to know that, once the compilation has completed, the tree will store a list of each of the polygons contained within its leaves. Once the BSP tree has been built, we need to write the polygon information to file in a format that is easy for the application to retrieve. Due to the fact that our application is already capable of loading mesh and polygon data from the IWF file, it makes sense that we export the resulting BSP tree polygon data out to file as a standard mesh. This will prevent us from having to implement another series of polygon import functionality and will allow the BSP tree's polygon data to be loaded seamlessly by any IWF import procedure."

This behavior can be observed in the function code outlined here. Should the BSP tree have been compiled, we take the polygon data contained within the compiled tree object and convert it into a 'CMesh' object such that it is compatible with both the standard mesh saving and loading procedures. This mesh is then added to the IWF file object in a manner that will cause this mesh to **always** be written to the file as the first mesh encountered during any subsequent import operation. Each mesh that follows this first mesh will be the detail objects that have not been processed or compiled into the BSP tree by this application. Moving forward it will be important to bear this in mind as we begin to develop the rendering application that will rely on this exported IWF file.

With all the various pieces of information stored within the compiler, it may seem as if the process involved in exporting the resulting scene data could be a little daunting. In fact, this process is relatively simple as we will discover when we begin to examine the implementation of the 'SaveScene' function below.

```
bool CCompiler::SaveScene( LPCTSTR FileName )
{
    ULONG       i;
    CFileIWF    IWFFile;
    CMesh      *pTreeMesh = NULL;

    try
    {
        // Fill up the IWF's internal structures
        IWFFile.m_vpTextureList  = m_vpTextureList;
        IWFFile.m_vpMaterialList = m_vpMaterialList;
        IWFFile.m_vpShaderList   = m_vpShaderList;
        IWFFile.m_vpEntityList   = m_vpEntityList;
```

This function accepts only a single parameter to which the application should pass the path and filename to the file in which it would like the current scene data to be saved. As with the loading operation undertaken in 'CompileScene', this function makes use of the 'CFileIWF' class to provide much of the

core file handling and export functionality. All we really need to do at this point is to populate the data structures within this class that it will then proceed to interpret and export to the final IWF file. To this end we begin by creating a local instance of this class in the form of the 'IWFFile' object shown at the top of this function.

Recall in our previous discussion of the 'CompileScene' function, we took ownership of several pieces of the imported data simply by performing a shallow copy on the STL vectors contained within the IWF loading class. During this save operation we can do exactly the same thing in reverse. By assigning the 'IWFFile' object's STL vector members to those matching vectors stored within the compiler object, we populate those data structures automatically. However, due to the fact that we must perform additional processing on the mesh information that is to be exported, we can only do this with the static information that was not modified during the compilation process. These include the texture, material, shader and entity vector members demonstrated in the previous code snippet.

With these other four STL vectors now populated, this function then proceeds to construct the all important mesh and polygon data that must also be exported to the IWF file. Depending on whether or not the application chose to enable the BSP tree compilation step, there are clearly two different situations that can arise. Either a BSP tree object has been created, from which we must populate the export file, or no tree was compiled and we simply need consider those meshes contained within the compiler's mesh array. The first of these two cases is shown below. If a BSP tree has been constructed at this point, the function will proceed to generate a compatible 'CMesh' object from the polygon data contained in the BSP tree. We achieve this by calling the 'BuildFromBSPTree' method of the 'CMesh' class covered earlier. To this function we pass the primary instance of our BSP tree class stored within the 'm_pBSPTree' member variable. Once completed, this newly generated mesh is then added to the IWF object's mesh list in exactly the same fashion as if this had been a standard mesh to begin with. Of course, the additional tree information such as the nodes and leaves will not be transferred into this mesh object. This additional information will be handled separately, as we shall see later on in this function.

```
        // If there is a tree, build a mesh from it
        if ( m_pBSPTree )
        {
            // Allocate a new tree mesh
            pTreeMesh = new CMesh;
            if (!pTreeMesh) throw std::bad_alloc();

            // Build the mesh from the bsp tree
            if (!pTreeMesh->BuildFromBSPTree( m_pBSPTree ))
                throw BCERR_OUTOFMEMORY;

            // Add the mesh to the file export list
            IWFFile.m_vpMeshList.push_back( pTreeMesh );

        }
```

The alternate case is executed when the application chose not to construct a BSP tree from the imported mesh data. In this case we simply add each of the **non-detail** meshes directly to the end of the 'IWFFile' object's mesh list ready for export.

```
        else
        {
            // Add any conventional meshes
            for ( i = 0; i < m_vpMeshList.size(); i++ )
            {
                CMesh * pMesh = m_vpMeshList[ i ];
                if ( !pMesh ) continue;

                // Add this if it's not a detail mesh
                if ( !(pMesh->Flags & MESH_DETAIL) )
                    IWFFile.m_vpMeshList.push_back( pMesh );

            } // Next Mesh

        } // End if No Tree
```

Although it may seem strange that we have chosen to add only those non-detail mesh objects to the export list in the previous case, remember that we must export any detail objects back out to file regardless of whether or not we have compiled a BSP tree. As a result, the detail objects are added to the list outside of the conditional if/else statement to ensure that these detail mesh objects are exported in both cases.

```
        // Loop through and add any detail meshes
        for ( i = 0; i < m_vpMeshList.size(); i++ )
        {
            CMesh * pMesh = m_vpMeshList[ i ];
            if ( !pMesh ) continue;

            // Add this if it's a detail mesh
            if ( pMesh->Flags & MESH_DETAIL )
                IWFFile.m_vpMeshList.push_back( pMesh );

        } // Next Mesh
```

Now that each of the 'IWFFile' object's four internal STL vectors have been populated with the appropriate data items, we can now call the 'Save' function to have that data written to file. To this function we pass both the name of the file to which the scene is to be exported, in addition to a pointer to the BSP tree object stored in the compiler's 'm_pBSPTree' member variable. While the purpose of the first parameter should be obvious, we pass the BSP tree object to this function as the second parameter in order for the IWF class to access and export the additional information stored within the tree. These include the nodes, leaves and plane information that may have been generated during the compilation process. In the next lesson, the tree object will also store portal and visibility information that the 'CFileIWF' class will also be responsible for retrieving and exporting.

Assuming the save operation was a success; we must now clear up any items that may have been allocated during this function's execution. The first of these is the mesh pointed to by the local 'pTreeMesh' variable. Recall that if a BSP tree object was constructed during the compilation process, this function was required to generate a new mesh object that could be added to the standard mesh list within the 'IWFFile' object. If this case has occurred, then we must release this temporary mesh object before exiting the function in order to prevent a memory leak. Finally we clear each of the 'IWFFile'

object's internal STL vector items. We must do this to prevent the objects stored within this array from being released prematurely when the 'IWFFile' object goes out of scope as this function returns.

```
        // Export file
        IWFFile.Save( FileName, m_pBSPTree );

        // Clean up
        if ( pTreeMesh ) delete pTreeMesh;
        IWFFile.m_vpMeshList.clear();
        IWFFile.m_vpTextureList.clear();
        IWFFile.m_vpShaderList.clear();
        IWFFile.m_vpEntityList.clear();
        IWFFile.m_vpMaterialList.clear();

    } // End Try Block
```

This function traps any errors encountered using a try/catch block mechanism. Because an exception may be thrown during either the IWF export procedure, or during any of the STL operations we perform in this function, the catch block below attempts to safely release any allocated memory in an identical manner to that outlined in the earlier code. If an exception was thrown within this function then the code in this catch block will be executed before returning a value of 'false' to the calling function. If no exception is thrown then the catch block is skipped and the function will return a value of 'true' denoting a successful export.

```
    catch (...)
    {
        if ( pTreeMesh ) delete pTreeMesh;
        IWFFile.m_vpMeshList.clear();
        IWFFile.m_vpTextureList.clear();
        IWFFile.m_vpShaderList.clear();
        IWFFile.m_vpEntityList.clear();
        IWFFile.m_vpMaterialList.clear();
        return false;

    } // End Catch Block

    // Success!!
    return true;
}
```

CCompiler::Release / ~CCompiler()

With the scene now fully processed, the only two functions that remain for us to examine are those responsible for cleaning up any objects or resources allocated during the import and compilation procedures. The first of these two functions is the 'Release' method. Similar to the functions of this type that we have implemented in the past, this can be called by the application should it wish to reuse any particular instance of the compiler class in a subsequent compilation task.

There is nothing really deep or complex about this function, the majority of which is dedicated to simply looping through each of the STL vectors maintained by the compiler class, and releasing any data storage objects stored in each element. These include meshes, textures, materials, shaders and entities.

Once each of these objects has been correctly released, the STL vectors in which they were contained are cleared in order to ensure that these arrays are cleared of the object pointers that are now invalid.

```
void CCompiler::Release()
{
    unsigned long i;

    // Destroy any loaded meshes
    for ( i = 0; i < m_vpMeshList.size(); i++ )
    {
        if ( m_vpMeshList[i] ) delete m_vpMeshList[i];

    } // Next Mesh

    // Destroy any loaded textures
    for ( i = 0; i < m_vpTextureList.size(); i++ )
    {
        if ( m_vpTextureList[i] ) delete m_vpTextureList[i];

    } // Next Texture

    // Destroy any loaded maerials
    for ( i = 0; i < m_vpMaterialList.size(); i++ )
    {
        if ( m_vpMaterialList[i] ) delete m_vpMaterialList[i];

    } // End If

    // Destroy any loaded shaders
    for ( i = 0; i < m_vpShaderList.size(); i++ )
    {
        if ( m_vpShaderList[i] ) delete m_vpShaderList[i];

    } // End If

    // Destroy any loaded entities
    for ( i = 0; i < m_vpEntityList.size(); i++ )
    {
        if ( m_vpEntityList[i] ) delete m_vpEntityList[i];

    } // End If

    // Clear the vectors
    m_vpMeshList.clear();
    m_vpTextureList.clear();
    m_vpMaterialList.clear();
    m_vpShaderList.clear();
    m_vpEntityList.clear();
```

With each of the scene data items released and cleared, the final task is to release the duplicated file name string contained in the ‘m_strFileName’ variable and to release the compiled BSP tree object should it exist.

```
    // Release strings
```

```
    if ( m_strFileName ) free( m_strFileName );
    m_strFileName = NULL;

    // Destroy any compiled BSP Tree
    if (m_pBSPTree) delete m_pBSPTree;
    m_pBSPTree = NULL;
}
```

The second of these two functions is the standard compiler class destructor. As has been the case in many of the classes we have encountered, this destructor simply calls the 'Release' function to ensure that no resources are leaked when the compiler class is destroyed or goes out of scope.

```
CCompiler::~CCompiler()
{
    // Clean up after ourselves
    Release();
}
```

With the completion of these two functions, we have reached the conclusion of the API exposed by the compiler class as it will commonly be used by any front end application. We have covered each task including the importing of the scene, preparing that scene data for processing, creating and executing each compiler module process and finally saving the data back to file. What remains to be covered is of course each of the compiler process classes that perform the pre-process tasks available in this lab project.

Adding BSP Tree Support – CBSPTree.cpp

Before we can realistically move on to discussing the compiler module classes, we must first discuss the integration of the new polygon-aligned BSP compiler supported into this lab project. Although this class is a compilation module in its own right, much of the functionality contained within this class is also used by other processing modules. In the case of the hidden surface removal process – which is actually executed *before* the main BSP compile step – we must create and compile individual BSP tree objects from each of meshes that we intend to merge before we can perform the union operation. Since this class is a major pre-requisite in many cases, it will be beneficial to us if we step outside of the program flow for a while and examine the entire set of BSP tree classes and compilation procedures.

A Note Regarding Linked Lists

The various compiler classes that we have previously implemented have made extensive use of the STL 'list' template class to provide much of the linked list support required by those applications. In this new compiler application we use manually constructed linked lists almost exclusively. We have encountered these concepts many times in previous lessons, an example of which would be the sibling pointer and iteration logic used extensively in our coverage of the D3DXFRAME and D3DXMESHCONTAINER structures. We have chosen to use manual linked lists in this application due to the many varied situations in which they are used requiring a high degree of flexibility not easily implemented when using the STL list template class.

With that in mind, let us now move on to discuss each of the supporting classes required by the BSP tree compilation process.

The CBSPFace Class

Up until this point we have been using the renderable 'CFace' class during the manipulation of our scene data. This class maintains those pieces of information that our rendering application may require – such as the texturing, material and surface normal data imported from the source IWF file. Recall that the 'CFace' class is also derived from a base of 'CPolygon' that provides much of the vertex management we require such as the 'Split' function we encountered earlier. Due to the fact that we want to compile the same information that we will ultimately be rendering in our run-time application, this class is derived directly from the 'CFace' class. By extending the renderable polygon class in this way, we automatically inherit the variables and functionality exposed by this base class without having to explicitly redeclare such members.

The 'CBSPFace' class is used exclusively by the BSP tree compilation process, and contains several additional member variables that are designed specifically for those tasks undertaken by the BSP tree class.

```
class CBSPFace : public CFace
{
public:

    // Constructors & Destructors
    CBSPFace( );
    CBSPFace( const CFace * pFace );

    // Public Variables for This Class
    bool        UsedAsSplitter;
    long        OriginalIndex;
    bool        Deleted;
    long        ChildSplit[2];
    CBSPFace   *Next;
    long        Plane;

    // Public Virtual Functions for This Class
    virtual HRESULT Split( const CPlane3& Plane, CBSPFace * FrontSplit,
                           CBSPFace * BackSplit, bool bReturnNoSplit = false );

};
```

Let us briefly examine each of the new member variables declared within this extended polygon class.

bool UsedAsSplitter

In the previous lab project 16.1 in which we developed a BSP node tree compiler class, the polygons were removed from consideration and attached to each node at the point of being selected as a separating plane candidate. During the compilation of the leaf based BSP tree however, we are required to pass any surviving polygon fragments through the tree in order for them to be collected in the attached leaf structures that will be created at the terminal nodes. While this could be achieved by

passing the polygons through the tree after the nodes and leaves have been constructed, the most efficient approach is to perform the polygon splitting and classification at the same time as building the tree itself. To this end, we need to be able to determine if a polygon fragment has already been chosen as a node plane candidate in order to remove it from consideration as it is passed through the tree during construction.

The 'UsedAsSplitter' variable is the means by which this is achieved. By testing the value of this member in the splitter selection step, we can determine whether a polygon has already been used to generate a node at a point further up in the recursive building process. With a default value of 'false', each polygon passed into the BSP tree class for compilation can initially be chosen for node plane candidacy. Whenever a polygon has been used in the creation of a node however, this member variable should be set to 'true'. Thus, whenever that polygon is encountered by the splitter selection routine at a point further down the tree, it would not be selected for a second time. This leaves us free to pass as many polygons as we need through the recursive BSP compilation function, creating a leaf only when there are no remaining polygons available for use as splitters in the polygon list specified.

long OriginalIndex
When used in conjunction with the more traditional split type BSP tree compilation process, this variable is used to transfer the index of this polygon's final location within the BSP trees centralized polygon array for storage in the leaf structure. When the application requests that the BSP tree compiler build a structure in which the leaves reference the original unsplit versions of the polygon data however – the non-split resulting tree type – this variable is used to track the index of the original polygon from which this particular fragment was created. Whenever a polygon is split, this original index value is copied into each of the resulting fragments. This enables us to identify the polygon from the original imported data set when any polygon fragment reaches a terminal node. We will explore this topic in greater depth a little later.

bool Deleted
During the hidden surface removal process, there are situations in which a polygon may have been split into multiple fragments of which none have been removed under the clipping rules specified. We will discover shortly how it is possible for us to easily repair these unnecessary splits and restore the polygon to its initial state. To do this however, requires that we do not physically delete the original polygons whenever they are found to span a BSP tree node that subsequently caused it to be split. For this reason, we maintain this 'Deleted' variable that we use to notify the HSR clipping procedure that it should simply be ignored as if it had been removed altogether. This allows us to simply set this variable to 'false', and those of any resulting split fragments to 'true', in order to undo any unnecessary split operations that may have occurred.

long ChildSplit[2]
Used only by the hidden surface removal support functionality provided by the BSP tree class, each element in this array stores indices to the two resulting polygon fragments added to the resulting polygon list should this polygon be split against a node's separating plane. We will examine how this variable is applied when we come to discuss these supporting functions later in this chapter.

CBSPFace * Next

This pointer variable is used to describe the *next* 'CBSPFace' object contained within any individual linked list being maintained by the compiler. This variable is used in several situations, each of which will be explained when they are encountered.

long Plane

Earlier in this chapter we discussed how we might build an array of planes that could be referenced by the nodes of the tree using a simple index. This member variable contains an index that references an element within such a plane array that will later be constructed within the BSP tree class. Recall that using a combined plane array allows us to reduce the amount of memory and file resources required to store the plane information which might otherwise be duplicated for coplanar nodes and polygons. In addition to reducing the memory footprint of this plane data, we will also shortly discuss how the accuracy of the BSP compilation process can be greatly improved by building shared and reusable plane data in this way.

CBSPFace::CBSPFace()

There are two constructors provided by this class. The first of these is the default class constructor shown below.

```
CBSPFace::CBSPFace()
{
    // Initialise anything we need
    UsedAsSplitter  = false;
    OriginalIndex   = -1;
    Next            = NULL;
    Deleted         = false;
    ChildSplit[0]   = -1;
    ChildSplit[1]   = -1;
    Plane           = -1;
}
```

As you can see, this constructor simply initializes each of the extended class member variables with a sensible default value. The only member whose default value is of any real importance here is the 'UsedAsSplitter' variable. Due to the fact that every polygon used in the construction of the BSP tree should be used as a node plane candidate at some point in the process, it is imperative that this variable is defaulted to 'false'. Remember that in C++, each of the base class constructors will be automatically executed in order. As a result, it is not necessary for us to default those member variables declared by the parent 'CFace' or 'CPolygon' classes here.

The second constructor defined by 'CBSPFace' class is an overload that accepts a single parameter of a 'CFace' object pointer type. This constructor is used in cases where the data contained within the polygon passed to this constructor should be duplicated into the new 'CBSPFace' polygon being constructed. This function begins in a similar manner to that of the default constructor by initializing the extended member variables declared by this class.

```
CBSPFace::CBSPFace( const CFace * pFace )
{
    // Initialise anything we need
    UsedAsSplitter  = false;
```

```
    OriginalIndex   = -1;
    Next            = NULL;
    Deleted         = false;
    ChildSplit[0]   = -1;
    ChildSplit[1]   = -1;
    Plane           = -1;
```

Once the ‘CBSPFace’ member variables have been set to their initial state, this function then proceeds to duplicate the information contained within the ‘CFace’ object passed to this constructor. As demonstrated below, this is achieved by first copying the values from those member variables declared by the ‘CFace’ class from the polygon object passed, into the corresponding members inherited by this object. Once this information has been transferred, the constructor then proceeds to duplicate the vertex data contained within the specified polygon object into the internal vertex array of the new polygon object being constructed.

```
    // Duplicate required values
    Normal          = pFace->Normal;
    TextureIndex    = pFace->TextureIndex;
    MaterialIndex   = pFace->MaterialIndex;
    ShaderIndex     = pFace->ShaderIndex;
    Flags           = pFace->Flags;
    SrcBlendMode    = pFace->SrcBlendMode;
    DestBlendMode   = pFace->DestBlendMode;

    // Duplicate Arrays
    AddVertices( pFace->VertexCount );
    memcpy( Vertices, pFace->Vertices, VertexCount * sizeof(CVertex));
}
```

CBSPFace::Split

Just as with the ‘CFace' class from which ‘CBSPFace’ is derived, we must provide an overloaded ‘Split’ function that is responsible for copying any additional member variables maintained by this class into each of the two split fragments. Notice that at the very start of this function we make a call to the base class’ implementation of the ‘Split’ procedure. Recall that in this case, the base from which this polygon class is derived is ‘CFace’ which is in turn derived from ‘CPolygon’. By calling the ‘CFace’ implementation of the split function, which in itself also calls the ‘CPolygon’ implementation, we ensure that all of those pieces of information maintained by either class in the hierarchy are duplicated into the two resulting polygon fragments.

```
HRESULT CBSPFace::Split( const CPlane3& SplitPlane, CBSPFace * FrontSplit,
                         CBSPFace * BackSplit )
{

    // Call base class implementation
    HRESULT ErrCode = CFace::Split( SplitPlane, FrontSplit, BackSplit );
    if (FAILED(ErrCode)) return ErrCode;

    // Copy remaining values
    if (FrontSplit)
    {
```

```
        FrontSplit->UsedAsSplitter = UsedAsSplitter;
        FrontSplit->OriginalIndex  = OriginalIndex;
        FrontSplit->Plane          = Plane;

    } // End If

    if (BackSplit)
    {
        BackSplit->UsedAsSplitter = UsedAsSplitter;
        BackSplit->OriginalIndex  = OriginalIndex;
        BackSplit->Plane          = Plane;

    } // End If

    // Success
    return BC_OK;
}
```

In this function, we copy three main pieces of information into each of the resulting polygon fragments. The first of these is the 'UsedAsSplitter' member. It is important that each of the split fragments inherit the value stored in this member whenever a polygon is split during the BSP compilation process. If this were not the case then, whenever a polygon that has already been selected for node plane candidacy is split, each child fragment would subsequently cause a new node plane to be generated at some point underneath the node originally generated. While this would not necessarily harm the integrity of the BSP tree or the validity of the solid / empty leaf information, creating these additional nodes would be wasteful and would merely increase the time it would take to traverse the hierarchy.

Secondly, we copy the source polygon's 'OriginalIndex' member value into each fragment. Again this is only really important when the application has chosen to construct a non-split resulting BSP tree, but we copy this information regardless of the tree type. As mentioned earlier, this index is used as a reference to the original scene polygon that was initially used to create each 'CBSPFace' object used in the BSP compilation process. Whenever a polygon gets split into two new fragments, each of these resulting polygons should maintain that same 'OriginalIndex' value so that we can correctly process this information when we reach a terminal node.

Finally, we duplicate the plane index into each of the resulting front and back fragments. Since each of the fragments generated by the polygon splitting operation are absolutely **guaranteed** to be coplanar with the source polygon, both of these fragments can share the same plane structure referenced by this polygon.

The CBSPLeaf Class

In the previous implementations of our kD-tree, oct-tree and quad-tree classes, we made extensive use of the 'CBaseLeaf' class to attach various pieces of information to each terminal node. This information included items such as the axis aligned bounding box of the space contained within the leaf, in addition to storing references to any polygon fragments that were contained within that leaf node's interior. The leaf structure used by the new BSP tree compiler class bears a striking resemblance to that original leaf concept.

```
class CBSPLeaf
{
public:

    // Constructors & Destructors
             CBSPLeaf( );
    virtual ~CBSPLeaf( );

    // Public Variables for This Class
    std::vector<long>   FaceIndices;
    CBounds3            Bounds;

    // Public Member Functions Omitted
};
```

Although our previous implementation of the leaf class contained many more run-time specific member variables, the basic premise is the same. This class stores a list of every polygon that was collected at any terminal node by the hierarchy compilation process, in addition to the axis aligned bounding box information that has been so useful to us in the past.

Let us examine each of these member variables before we move on to discuss the methods exposed by this class.

std::vector<long> FaceIndices

Unlike the previous implementation of the 'CBaseLeaf' class in which we stored direct pointers to scene polygon objects, recall that we are using an index based, file friendly mechanism in order to reference the different pieces of information in the tree. The 'FaceIndices' member variable therefore, stores a list of indices that reference individual elements with the BSP tree class's internal polygon array. Because this array need only store simple integer values that do not change after the leaf has been built, this member uses a simple STL vector type to provide an easy means by which to insert and access the polygon index data. Once the tree has been built, the indices stored in each leaf can then be used to retrieve the relevant data from the BSP tree's centralized polygon array.

CBounds3 Bounds

The 'Bounds' member is of the type 'CBounds3' – one of our new math utility classes – and is used to store the world space axis aligned extents of every polygon collected and stored within this leaf. This information will be written out to the export file along with the leaf structure, and can be imported and used in a run-time / rendering application to provide additional frustum culling or broad phase information when collect leaves for further consideration.

CBSPLeaf::BuildFaceIndices

The 'BuildFaceIndices' method is the only function defined within the 'CBSPLeaf' class. This function is designed to populate the polygon index data contained within the 'FaceIndices' STL vector. This function is called only by the BSP compiler's recursive 'BuildBSPTree' function and accepts a single parameter that describes the first item in a linked list of polygons to be stored within this leaf. Prior to calling this function, the 'BuildBSPTree' function updates the 'OriginalIndex' member of each 'CBSPFace' object contained within this list. Each 'OriginalIndex' variable is assigned a value that

describes the final location within the BSP tree's internal polygon array into which the polygon has been inserted. This index information is subsequently used to populate the member 'FaceIndices' array within this function.

The first thing we do in this function is to reset the bounding box member that will be used to describe the world space extents of every polygon stored within this leaf. This bounding box will be constructed incrementally as each polygon within the specified linked list is visited later in this function.

```
bool CBSPLeaf::BuildFaceIndices( CBSPFace * pFaceList )
{

    CBSPFace       *Iterator      = pFaceList;
    unsigned long   OriginalIndex = 0;

    // Reset bounds, will be rebuilt
    Bounds.Reset();
```

With the 'Bounds' member reset, we can now begin to iterate through the specified polygon list. Notice in the previous code snippet how the pointer describing the first element of the list is copied into a local variable named 'Iterator'. This variable will be used as the means by which we iterate through the linked polygon list.

While traversing the linked list of polygon data passed to this function, the first thing we must do is extract the 'OriginalIndex' value from the current polygon that will be added to the 'FaceIndices' member array. Before this happens however, this function includes a test to determine whether or not this index has already been added to the leaf's internal polygon list. This is achieved with a call to the 'std::find' library function that is designed to iterate through an STL vector, searching for an element that contains the specified value. In doing so, we prevent duplicate polygon indices from being added to the same leaf for whatever reason.

If no duplicate was found in the 'FaceIndices', we are now free to add the polygon index to this array. Due to the fact that we are adding many potential polygon indices to the STL vector member within a loop however, we use the capacity threshold resizing logic in order to prevent this array from being resized / reallocated for every index added. We have observed this kind of vector resizing logic in the past, but to summarize the process used here; if the total number of elements in the vector *including* the new index is found to exceed its current internal capacity then we allocate additional space by calling the vector's 'Reserve' method. To this method we specify a capacity equal to the current number of items stored in the vector plus an additional number of slack elements into which the vector can grow. In this way, the vector will only ever be physically resized each time we have added a number of indices equal to this additional slack space. The number of additional elements reserved in this function is defined by the constant shown below.

```
#define BSP_ARRAY_THRESHOLD 100
```

Once we have reserved any required number of elements in the 'FaceIndices' member, we can then add the 'OriginalIndex' value taken from the current polygon with a call to the STL vector's 'push_back' method.

```
    // Iterate building list
    while ( Iterator != NULL )
    {
        // Get the original polygon's index
        OriginalIndex = Iterator->OriginalIndex;

        try
        {
            // Make sure this index does not already exist in the array?
            if ( std::find(FaceIndices.begin(), FaceIndices.end(),
                           OriginalIndex) == FaceIndices.end())
            {

                // Resize the vector if we need to
                if (FaceIndices.size() >= (FaceIndices.capacity() - 1))
                {
                    FaceIndices.reserve( FaceIndices.size() +
                                         BSP_ARRAY_THRESHOLD );
                } // End If

                // Finally add this face index to the list
                FaceIndices.push_back(OriginalIndex);

            } // End If needs adding

        } // End Try Block
```

If an exception occurred during any of the above operations, we need to catch this situation and release any resources that were allocated before returning a failure code to the calling function

```
        catch (...)
        {
            // Clean up and bail
            FaceIndices.clear();
            return false;
        } // End Catch
```

In the final stages of this loop we must now grow the leaf's bounding box to include the current polygon's vertex data. This is done with a call to the 'CalculateFromPolygon' method of the leaf' 'Bounds' member. Recall that, in order for this function to grow the bounding box extent values using the specified vertex data, we must pass a value of false to the final 'reset' parameter.

With this polygon fully considered, we finally move on to the next polygon ready for the following iteration of the while loop. If this polygon was the last item in the linked list, the 'Next' variable will contain a value of NULL that would be subsequently transferred into the local 'Iterator' pointer, causing the while loop to exit.

```
        // Build into current bounding box
        Bounds.CalculateFromPolygon( Iterator->Vertices, Iterator->VertexCount,
                                     sizeof(CVertex), false );
```

```
        // Move to next poly
        Iterator = Iterator->Next;

    } // End While
```

At this stage, the 'FaceIndices' STL vector will contain an index to each polygon passed in to this leaf for storage. When adding each of the polygon indices to this array, recall that we used a capacity threshold technique to prevent the STL vector from having to be reallocated and duplicated every time a new index value was added. As we know, this block resizing technique can result in an STL vector having an internal capacity that is much larger than might actually be required to store the index values. In order to prevent these arrays from needlessly consuming this additional slack memory, the final task undertaken by this function is to optimize the memory allocated by the 'FaceIndices' vector if required.

```
    // Optimize the vector
    if (FaceIndices.size() < FaceIndices.capacity())
        FaceIndices.resize( FaceIndices.size() );

    // Success
    return true;
}
```

The CBSPNode Class

This class is used to represent the individual nodes within the hierarchy of the new BSP leaf tree. Similar to the node class we have previously implemented in both the kD-tree and BSP node tree compiler projects, this node class is represented by a single separating plane used to partition the space on either side into two equal halfspaces. This is a relatively simple class that is based on those we are already familiar with so let us move straight on to examining the class declaration.

```
class CBSPNode
{
public:

    // Constructors & Destructors
             CBSPNode( );
    virtual ~CBSPNode( ) {};

    // Public Variables for This Class
    long            Plane;
    long            Front;
    long            Back;
    CBounds3        Bounds;

    // Public Member Functions Omitted
};
```

At this point, it should be reasonably clear as to the purpose of each of the member variables declared within this class. In the interest of completeness however, let us just briefly examine each member before moving on to discuss the class methods.

long Plane
Just as with the 'CBSPFace' class, the node maintains an index into the BSP tree's internal array in order to describe the plane on which this node lies. This index is taken directly from the polygon object that was used as the candidate plane for the generation of this node. In this way we are able to reuse not only that plane information shared between individual polygons, but also between coplanar nodes and polygons alike.

long Front
This member variable has a dual purpose. Recall from our earlier discussion of the file based BSP data structures; we will use the same front member variable to attach both nodes *and* leaves to the appropriate side of this node. The value stored in this member is used to represent an index into one of the BSP tree's centralized data arrays. The type of data structure that is attached to the front of this node via this member depends on the sign of the value stored here. If this variable contains a negative value, this means that there is a leaf attached to the front of the node and is used to represent an index to an element within the BSP tree's leaf array. If this value is positive, then there is simply another node attached as the front child. This is then used as the index into the BSP tree's node array.

long Back
This purpose and use of this member variable is identical to that of the aforementioned 'Front' member. The only difference here is that this member is used to describe the index of either the node or leaf attached as the child *behind* this node.

CBounds3 Bounds
This member represents the axis aligned world space extents for this node. Recall that the bounding box contained within a BSP tree node is used to describe the world space extents of *every* polygon that will exist in the tree at some point beneath it in the hierarchy.

CBSPNode::CBSPNode()

This class contains only a single default constructor in which each of the primitive typed member variables are defaulted to a value of '-1'. This value is used to signify that the information has not yet been initialized. The only variable that we do not initialize here is the 'Bounds' member. This is due to the fact that the constructor of this object will be called implicitly, whenever an instance of this node class is created.

```
CBSPNode::CBSPNode()
{
    // Initialise anything we need
    Plane   = -1;
    Front   = -1;
    Back    = -1;
}
```

CBSPNode::CalculateBounds

The only member function defined within this class is the 'CaclulateBounds' method shown in the following block of code. Called during the BSP construction process, this is a utility function designed

to populate the ‘Bounds’ member of this object with the world space extents of every polygon that will exist in the tree at some point underneath this node.

This method accepts two parameters. The first of these two describes the first element in a linked list of ‘CBSPFace’ object pointers that will be taken into consideration during the calculation of this bounding box. The second parameter is a simple Boolean value that dictates whether the specified polygon data should be used to simply grow any existing bounding box values, or whether it should first be reset.

```
void CBSPNode::CalculateBounds( CBSPFace * pFaceList, bool ResetBounds )
{
    CBSPFace *Iterator = pFaceList;

    // Reset bounding box and grow from default state?
    if ( ResetBounds ) Bounds.Reset();

    // Calculate bounding box based on the face list passed
    while ( Iterator )
    {
        // Grow the bounding box values
        Bounds.CalculateFromPolygon( Iterator->Vertices, Iterator->VertexCount,
                                     sizeof(CVertex), false );

        // Move on to the next polygon
        Iterator = Iterator->Next;

    } // End If
}
```

The techniques used in the development of this function have been discussed in detail elsewhere in this lesson, so we will not go into any further detail here. We will discuss how and when this function is to be used shortly, as we continue on to discuss the main leaf based BSP tree compilation class in the following section.

The CBSPTree Class

Up until this point we have spent a great deal of time focusing on the specifics of our new pre-processing tool and the various associated framework classes that have been introduced. These discussions were vital in order for us to understand the techniques involved in building the compiler application for lab project 16.2. In addition, this coverage has laid the foundation that will allow us to discuss the various compilation modules, without having to stop and examine any new support and utility functionality utilized in the process. Let us therefore continue on to examine the new polygon aligned BSP leaf tree compiler that we have made reference to several times in this chapter.

While the ‘CBSPTree’ class serves as the basis for many of the compilation modules that are covered in both this lesson and the next, it also acts as a compilation module in its own right. Referring back to our discussion of the ‘PerformBSP’ method of the ‘CCompiler class, we know that a BSP tree is constructed from the scene information in much the same way as we have done in each of our previous spatial hierarchy lab project applications. This has traditionally involved passing scene polygon data into the compiler class, triggering the compilation process and finally interpreting and storing the compiled

information in a manner suitable for rendering. In this pre-processing application, the high level concepts involved in the use of the BSP tree class as a compilation module are very similar. The one exception is that, after compiling the tree we must interpret and store the compiled information in a manner suitable for export. With this in mind, let us now examine the declaration of the new 'CBSPTree' class in order to see just what information will be built.

As before, we have removed the list of member functions in this declaration to improve readability, so check the source code for a full list of methods.

```
class CBSPTree
{
public:

    // Constructors & Destructors for This Class.
             CBSPTree();
    virtual ~CBSPTree();

private:

    // Private Variables for This Class.
    CBSPFace       *m_pFaceList;
    unsigned long   m_lActiveFaces;
    vectorNode      m_vpNodes;
    vectorPlane     m_vpPlanes;
    vectorLeaf      m_vpLeaves;
    vectorBSPFace   m_vpFaces;
    vectorBSPFace   m_vpGarbage;
    CBounds3        m_Bounds;
    BSPOPTIONS      m_OptionSet;
    ILogger        *m_pLogger;
    CCompiler      *m_pParent;
};
```

As you can see, there are many variables declared within this new 'CBSPTree' class. While we are familiar with the types of information stored within many of these members at this point, there are several items in this list that we have not yet encountered. Therefore, before we move on to the discussion of the class implementation, let us first familiarize ourselves with the role of each of these member variables.

CBSPFace * m_pFaceList

Before we can even begin to think about compiling the tree structure, we obviously must have access to the list of polygons to which the nodes of the polygon-aligned BSP tree will be oriented. This member variable contains the first / head element within a linked list of 'CBSPFace' objects that will be used as the source set for the BSP compilation process. In the earlier discussion of the 'CCompiler::PerformBSP' function, we saw how each polygon in the scene was passed into the BSP tree object through a call to its 'AddFaces' method. This function accepted an array of 'CFace' object pointers that were to be considered as valid polygons for use by the BSP tree compilation process. We will discuss how the 'AddFaces' method is implemented shortly, but in summary this function takes each of the specified 'CFace' objects, duplicates that polygon's data into a new 'CBSPFace' object before finally attaching that new polygon to the linked list maintained by this member.

vectorNode m_vpNodes
The ‘m_vpNodes’ member is used as the centralized container array for the *final* list of hierarchy nodes contained within the compiled BSP tree. The information contained within this node array will later be exported to file after each compilation process has been run. This member variable is of the type ‘vectorNode’ which is a typedef of a standard STL vector template designed to store a series of ‘CBSPNode’ object pointers.

```
typedef std::vector<CBSPNode*>   vectorNode;
```

In order to access the data stored within this vector easily and safely, the BSP tree class defines both the ‘GetNode’ and ‘GetNodeCount’ accessor functions for use by the application. Existing elements can also be assigned with the use of the ‘SetNode’ method. A list of these accessor functions can be found and the end of this variable listing if you wish to examine them.

vectorPlane m_vpPlanes
This member variable is intended to store a list of every plane used by both the polygons and nodes contained within this BSP tree. Recall that we will be relying heavily upon shared plane information in this BSP tree implementation in order to save space and – as we will discover shortly – to improve the accuracy of the BSP compilation process. As a result, individual planes stored within this array may be referenced by many different sources before, after and during the compilation process. The information contained within this plane array will also later be exported to file after each compilation process has been run.

The ‘m_vpPlanes’ member is of the type ‘vectorPlane’ which is a typedef of a standard STL vector template designed to store a series of ‘CPlane3’ object pointers.

```
typedef std::vector<CPlane3*>    vectorPlane;
```

In order to access the data stored within this vector easily and safely, the BSP tree class defines both the ‘GetPlane’ and ‘GetPlaneCount’ accessor functions for use by the application. Existing elements can also be assigned with the use of the ‘SetPlane’ method.

vectorLeaf m_vpLeaves
The ‘m_vpLeaves’ member is used as the centralized container array for the *final* list of leaf objects contained within the compiled BSP tree. The information contained within this leaf array will later be exported to file after each compilation process has been run. This member variable is of the type ‘vectorLeaf’ which is a typedef of a standard STL vector template designed to store a series of ‘CBSPLeaf’ object pointers.

```
typedef std::vector<CBSPLeaf*>   vectorLeaf;
```

In order to access the data stored within this vector easily and safely, the BSP tree class defines both the ‘GetLeaf’ and ‘GetLeafCount’ accessor functions for use by the application. Existing elements can also be assigned with the use of the ‘SetLeaf’ method.

vectorBSPFace m_vpFaces
The 'm_vpFaces' member is used as the centralized container array for the *final* list of polygon objects contained within the compiled BSP tree. The information contained within this polygon array will later be exported to file after each compilation process has been run. This member variable is of the type 'vectorBSPFace' which is a typedef of a standard STL vector template designed to store a series of 'CBSPFace' object pointers.

```
typedef std::vector<CBSPFace*>   vectorBSPFace;
```

In order to access the data stored within this vector easily and safely, the BSP tree class defines both the 'GetFace' and 'GetFaceCount' accessor functions for use by the application. Existing elements can also be assigned with the use of the 'SetFace' method.

vectorBSPFace m_vpGarbage
Logically, as the tree compilation process becomes ever more comprehensive, there are likewise more situations in which failures might occur. If a failure does happen, it is always preferable for the application to gracefully handle this situation such that the user might take further action to resolve the source of the problem. This member array is provided in order to make the process of handling such errors a little easier during the recursive BSP compilation procedure. By providing a convenient location for the collection of polygons that should be released at a later point in time, we can actually bypass some significant problems introduced when trying to clean up any allocated polygon data in the middle of this heavily recursive process. The specifics of the error handling and garbage collection scheme will be covered a little later on in this section.

CBounds3 m_Bounds
As with many of the classes to which we have previously been introduced, the BSP tree class also stores an axis aligned bounding box that is used to represent the total world space extents of every node, leaf and polygon contained within the compiled tree. This information will be crucial during the portal compilation process we will develop in the next lesson.

unsigned long m_lActiveFaces
This member is used primarily for the purposes of informing the user about the progress of the compilation process. The value stored here simply maintains a count of the number of polygons currently active within the compilation process. With this information we can determine how far through the BSP compile process we have come simply by comparing this total number of polygons against the number of polygons that have been removed from consideration at each step.

BSPOPTIONS m_OptionSet
This member stores the various settings, specified by the application to inform the compiler how the BSP tree should be compiled. This options structure is set using the 'SetOptions' accessor function outlined on the following page.

ILogger * m_pLogger
In order for the BSP compiler to report progress, error and useful status information to the user, this member variable stores a pointer to the application defined logging class. While the logging feature remains optional, this member can be set with a call to the 'SetLogger' accessor function defined by this

class. In this lab project, the logging class instance is passed to this object by the 'CCompiler::PerformBSP' function.

CCompiler * m_pParent

Each compilation module class maintains a member that stores a reference to the parent compiler class that created it. This member is set with a call to the 'SetParent' accessor function defined by each of these classes. This information is used by each of these processing modules in order to test the current state of the compiler to determine if the compile operation has been either paused or cancelled. Although we have omitted much of the logging and progress functionality from the code listings in this workbook, you should take a look at the source code for lab project 16.2 to see this process in action.

BSP Tree Accessor Functions

There are many accessor functions defined within the BSP tree class. In each case, the body for these functions can be found in the class declaration contained in the header file, rather than in the source module / .CPP file. We include the function bodies in the header mostly for convenience. However, because these functions are very simple and are accessible by any source module that might include this header, these functions are much more likely to be chosen to be inline functions. Due to the fact that these methods are so small, and will be called often, this is an ideal situation.

We will not go into detail about how each of these functions is implemented, for this you should take a look at the 'CBSPTree.h' header file contained in the 'Compiler Source' project directory. Instead we will give a brief overview of each category of accessor function available in this class.

The first series of accessor functions defined within the BSP tree class are those which return the number of elements stored within the STL vector for each type of data stored. These functions wrap a call to the STL vector 'size' method, and simply return that information to the calling function.

```
unsigned long   GetNodeCount  ( ) const
unsigned long   GetLeafCount  ( ) const
unsigned long   GetPlaneCount ( ) const
unsigned long   GetFaceCount  ( ) const
```

The second set of accessor functions defined here are those that actually retrieve a pointer to each specific data class from the member arrays. Each of these functions accepts a single parameter that is used to specify the index to the element in the array which the calling function would like to retrieve. Rather than simply have the underlying STL vector throw an exception if the specified index is out of bounds however, these functions test the value of the index parameter, returning NULL if it exceeded the end of the array. This allows the calling function to retrieve each type of object safely without having to wrap each call with exception handling logic.

```
CBSPNode       *GetNode       ( unsigned long Index ) const
CPlane3        *GetPlane      ( unsigned long Index ) const
CBSPLeaf       *GetLeaf       ( unsigned long Index ) const
CBSPFace       *GetFace       ( unsigned long Index ) const
```

The third set of accessor functions are called in order to *set* an individual element in each of the member arrays. These functions each accept two parameters. The first is the index to the element in the relevant

array that the calling function would like to set. The second is a pointer to the data object that should be stored in that specified array element.

```
void            SetNode       ( unsigned long Index, CBSPNode   * pNode  )
void            SetPlane      ( unsigned long Index, CPlane3    * pPlane )
void            SetLeaf       ( unsigned long Index, CBSPLeaf   * pLeaf  )
void            SetFace       ( unsigned long Index, CBSPFace   * pFace  )
```

The fourth and final set of accessor functions are those that set the various pieces of information required by each compilation process module. These include the process options, a pointer to the parent compiler and a pointer to the class defined logging class. These functions simply store those specified values into the applicable member variables declared by this class.

```
void            SetOptions    ( BSPOPTIONS Options    )
void            SetParent     ( CCompiler * pCompiler )
void            SetLogger     ( ILogger * pLogger     )
```

Now that we have familiarized ourselves with each component of the ‘CBSPTree’ class declaration, let us move directly on to the implementation of the all important class methods.

CBSPTree::AddFaces

The ‘AddFaces’ method is the means by which any applicable scene polygon data is added to the tree ready for compilation. We have seen how this method is called in our earlier coverage of the ‘CCompiler::PerformBSP’ function.

```
HRESULT CBSPTree::AddFaces( CFace ** ppFaces, unsigned long lFaceCount )
{
    CBSPFace *NewFace = NULL;

    // Validate Parameters
    if ( !ppFaces || lFaceCount <= 0 ) return BCERR_INVALIDPARAMS;

    // Release the old tree data (if any)
    if (!m_pFaceList) ReleaseTree();
```

This function accepts two parameters. The first is an array of ‘CFace' object *pointers* that contains the list of polygons that have been selected by the application to be included in the BSP compilation process. The second parameter specifies the number of polygons contained within that array.

After ensuring that each of the specified parameters are valid, this function first releases any tree data that already exists within this object should the class have been previously used. This is only done during the first call to the ‘AddFaces’ function however. We can determine if this is the first time that this function has been called by testing the value stored within the ‘m_pFaceList’ member variable. If this variable currently contains a NULL pointer, this means that no polygon data has yet been collected. This is something that will no longer be the case after this function has completed its first execution.

At this stage, we begin to loop through each of the polygons passed into this function ready to populate the ‘m_pFaceList’ linked list member variable. This list will contain the data that is actually used to

construct the BSP tree in the next step. To this end, each valid ‘CFace' polygon object encountered in the array passed to this function is duplicated into a newly allocated ‘CBSPFace’ polygon with a call to the ‘AllocBSPFace’ function. This function wraps much of the allocation and error handling logic on our behalf and is an easy and safe way to create a new BSP specific polygon object. Into this newly duplicated polygon we store the index that describes the order in which this polygon was created. This value is retrieved from the ‘m_lActiveFaces’ member variable which is subsequently incremented.

```
    // First add the mesh polygons to the initial linked list
    for ( unsigned long i = 0; i < lFaceCount; i++ )
    {

        // Add if available
        if (ppFaces[i])
        {
            // Allocate the new BSPFace
            if (!(NewFace = AllocBSPFace( ppFaces[i] )))
            {
                ReleaseTree();
                return BCERR_OUTOFMEMORY;
            } // End If Out of Memory

            // Store original index for later use
            NewFace->OriginalIndex = m_lActiveFaces;
            m_lActiveFaces++;
```

With this source polygon duplicated into the new ‘CBSPFace’ typed object, we attach this new polygon to the head of the internal source polygon linked list referenced by the ‘m_pFaceList’ member.

```
            // Attach this to the list
            NewFace->Next = m_pFaceList;
            m_pFaceList   = NewFace;
```

In the spatial hierarchy concepts developed in previous lab projects, we had the ability to generate trees in which the polygon data stored in its leaves were not split against the separating planes. When the non-split option was selected in these earlier compiler classes, the source polygon data used in the construction of the tree remained in the tree’s internal polygon array from the start. This was in contrast to the splitting method in which the polygon fragments were added to the internal polygon array only when they were being added to a leaf in the tree. This same logic is applied to our new BSP leaf tree compiler.

In the following block of code we can see that the current original source polygon is duplicated and added directly to the tree’s final polygon array if the non-split option was selected by the application. Recall that earlier in this function we stored the current face count in the ‘OriginalIndex’ member of each polygon stored in the linked list. If the non-split resulting tree option has been selected, this ‘OriginalIndex’ value will correspond with the element in which these duplicate polygons have been stored in the tree’s final polygon array. We will examine how all of this is tied together in the coverage of the ‘ProcessLeafFaces’ method later in this section.

```
            // Non Split tree?
            if ( m_OptionSet.TreeType == BSP_TYPE_NONSPLIT )
```

```
            {
                // Allocate some storage space
                if (!IncreaseFaceCount())
                {
                    ReleaseTree();
                    return BCERR_OUTOFMEMORY;
                } // End if out of memory

                // Allocate a duplicate of the original original
                if (!(NewFace = AllocBSPFace( ppFaces[i] )))
                {
                    ReleaseTree();
                    return BCERR_OUTOFMEMORY;
                } // End if out of memory

                // Store this new pointer
                SetFace( GetFaceCount() - 1, NewFace );

            } // End if backup required

        } // End If Face Available
```

With the current polygon now processed, we move onto the next item in the specified array. Once the loop has run its course, this function returns a success code back to the calling function.

```
    } // Next Face

    // Success
    return BC_OK;
}
```

CBSPTree::CompileTree

After having added each of the polygons to be considered by the BSP tree compiler, this function is called in order to begin the actual tree compilation process. Put simply, this method wraps each of the calls to the various pieces of functionality required in order to construct the BSP tree.

The first process that we must undertake when building the BSP tree in this application is to construct the combined plane array into which each ‘CBSPFace’ object’s ‘Plane’ member will reference. This information is used both to reduce the amount of memory consumed for plane data between coplanar polygons, in addition to improving the accuracy of the BSP compilation process as we shall demonstrate shortly.

Once the pre-compiled plane array information has been generated, we are ready to begin the construction of the BSP tree itself. Before we can do this however, we must first allocate the root node that will be passed into the initial call of the main recursive build procedure. This is done with a call to the ‘IncreaseNodeCount’ method discussed later in this section.

```
HRESULT CBSPTree::CompileTree( )
{
    HRESULT ErrCode;
```

```
    // Validate values
    if (!m_pFaceList) return BCERR_INVALIDPARAMS;

    // Calculate our plane array
    if (FAILED(ErrCode = BuildPlaneArray())) return ErrCode;

    // Allocate our root node
    if (!IncreaseNodeCount()) return BCERR_OUTOFMEMORY;

    // Log – BSP: Building BSP Tree
```

At this point, we make a call to the 'BuildBSPTree' function, passing in both the index to the root node we allocated a moment before (the index of the root node will always be zero in this implementation), in addition to the linked list of polygon data referenced by the 'm_pFaceList' member.

```
    // Compile the BSP Tree
    if (FAILED(ErrCode = BuildBSPTree(0, m_pFaceList))) return ErrCode;
```

Assuming no errors occurred during the compilation process above, we should now have compiled a fully functional BSP tree. With each of the polygons in the source linked list now stored within the BSP tree data structures, we no longer own these polygons. As a result, we reset the 'm_pFaceList' with a NULL pointer value to remove all references to those polygons should we attempt to release this list, or append new polygons to it with future calls to 'AddFaces'.

```
    // Reset Face List
    m_pFaceList = NULL;
```

Now that we have populated the internal polygon array with the final set used in the construction of the BSP tree, the final task undertaken by this function is to calculate the tree's overall bounding box. We can do this quickly and efficiently using the 'CalculateFromPolygon' method exposed by the 'm_Bounds' object, passing in the vertex data from each polygon now stored in the tree as a whole.

With this final piece of information built, we finally return back to the calling function, notifying it of our success.

```
    // Build the trees bounding box
    m_Bounds.Reset();
    for ( ULONG i = 0; i < GetFaceCount(); i++ )
    {
        m_Bounds.CalculateFromPolygon( GetFace(i)->Vertices,
                                       GetFace(i)->VertexCount,
                                       sizeof(CVertex), false );

    } // Next Face

    // Log – BSP: Compile Success

    // Success
    return BC_OK;
}
```

CBSPTree::BuildPlaneArray

We have discussed at several points within this lesson just how beneficial the construction of a combined plane array can be during the compilation of a polygon aligned BSP tree. Not only does this pre-building technique provide a much smaller memory and file footprint with respect to the plane data itself, it also allows us to construct a BSP compiler class that is extremely stable and as fault tolerant as possible. Regarding the latter, we will observe later how building this plane array in advance can eliminate hundreds of potentially error prone polygon and vertex classification tests to be replaced with a simple integer equality test. Before we get to that point however, let us first examine how this new combined plane array is constructed. This should give us a better understanding of how these methods employ this information later in this chapter.

```
HRESULT CBSPTree::BuildPlaneArray()
{
    CBSPFace * Iterator = NULL;
    CPlane3    Plane, * TestPlane;
    CVector3   Normal, CentrePoint;
    int        i, PlaneCount;
    float      Distance;

    // Log – BSP: Building Initial Plane Array
```

This function accepts no parameters and employs a simple HRESULT style return code system for reporting any errors that may have occurred.

Given the list of source polygons contained within the ‘m_pFaceList’ member, this function begins by looping through each of these polygons using the standard linked list iteration logic we should already be familiar with. For each polygon in the list, the first job is to construct a ‘CPlane3’ object that describes the plane on which this polygon lies. As we know, in order to construct a plane from a polygon we need two pieces of information. The first is the polygon normal and the second is a point that lies on the intended plane. For the point we might perhaps use the position of the first vertex stored within the polygon vertex array. Due to the limited accuracy of single precision floating point values however, there is no guarantee that every vertex in the polygon lies on *exactly* the same plane. In many cases there will be a very slight twist to the polygon due to the vertices becoming slightly misaligned. As a result, in such a accuracy sensitive situation, it is not really ideal that we select any one vertex as the point used to construct the plane. For this reason, the first thing we do inside the main loop is to sum the position of every vertex in the current polygon, and then divide by the total number of vertices. This will give us an average center point that should provide us with a more accurate point on which to base the plane.

With the surface normal taken directly from the polygon and the averaged center point now computed we construct a new plane object by passing the normal and point information to the relevant overloaded ‘CPlane3’ constructor. This will automatically convert the normal/point information into the normal/distance values maintained by the plane class.

```
    // Loop through each face
    for ( Iterator = m_pFaceList; Iterator; Iterator = Iterator->Next )
```

```
    {
        // Calculate polygons centre point
        CentrePoint = Iterator->Vertices[0];
        for ( i = 1; i < (signed)Iterator->VertexCount; i++ )
              CentrePoint += Iterator->Vertices[i];

        // Average Vertices
        CentrePoint /= (float)Iterator->VertexCount;

        // Calculate polygons plane
        Plane = CPlane3( Iterator->Normal, CentrePoint );
```

Once we have generated the plane based on the information contained within the current polygon, this function then proceeds to check if a similar plane already exists in the plane array constructed so far. We can do this simply by comparing both the normal and the distance values contained within each plane. If both planes are equal – remembering to use a tolerance in order to ensure that we cater for any floating point precision errors – then we simply break from the loop having found a matching plane.

```
        // Search through plane list to see if one already exists
        PlaneCount = GetPlaneCount();
        for ( i = 0; i < PlaneCount; i++ )
        {
            // Retrieve the plane details
            TestPlane = GetPlane(i);

            // Test the plane details
            if ( fabsf(TestPlane->Distance - Plane.Distance) < 1e-5f &&
                 Plane.Normal.FuzzyCompare( TestPlane->Normal, 1e-5f )) break;

        } // Next Plane
```

If the previous loop ran to its conclusion then we know that no plane matching that of the current polygon was found in the plane array. We can test for this case by checking to see if the loop counter variable ‘i' is equal to the number of planes that were to be tested in that loop – ‘PlaneCount’. If this is the case then this function proceeds to add the unique plane described by this polygon the plane array stored within this class.

Now that we have an index to either an already existing plane, or the newly generated plane, this index must finally be stored in the polygon’s ‘Plane’ member ready for use in the BSP compilation process.

```
        // Add plane if none found
        if ( i == PlaneCount )
        {
            if (!IncreasePlaneCount()) return BCERR_OUTOFMEMORY;
            *GetPlane( PlaneCount ) = Plane;

        } // End if no plane found

        // Store this plane index
        Iterator->Plane = i;
```

We mentioned earlier that it was possible for floating point inaccuracies to cause one more of the vertices in this polygon to be slightly misaligned, causing this polygon to twist to a small degree. Herein lies one of the techniques that we have implemented to help improve the accuracy of the BSP tree compiler.

In the following code we begin to loop through each of the vertices contained within the current polygon. For each vertex encountered here we first calculate the distance at which that vertex lies from the new or existing plane we assigned to the polygon in the previous steps. We then use this information to push the vertex back along the plane normal by that distance such that it will now lie exactly on the surface of that plane. As you might imagine, once every vertex in the polygon has been processed in this way, any twisting that may have caused incorrect classification results during compilation should now have been resolved.

With this step completed we then display the updated progress information to the user – via the logging interface – and move on to the next polygon.

```
        // Retrieve the plane details
        Normal   = GetPlane(i)->Normal;
        Distance = GetPlane(i)->Distance;

        // Ensure that all vertices are on the selected plane
        for ( unsigned long v = 0; v < Iterator->VertexCount; v++ )
        {
            float result = Iterator->Vertices[v].Dot( Normal ) + Distance;
            Iterator->Vertices[v] += (Normal * -result);

        } // Next Vertex

        // Log – BSP : Update Progress

    } // Next Face
```

Once every polygon in the ‘m_pFaceList’ linked list member has been tested and assigned a plane index, this function finishes simply by reporting the successful completion to the user, and returning control back to the calling function – ‘CompileTree’.

```
    // Log – BSP: Operation Successful

    // Success
    return BC_OK;

}
```

CBSPTree::BuildBSPTree

This recursive function is the very heart of the polygon aligned BSP leaf tree compiler. It is functionally very similar in many respects to those we have implemented in the past. From a high level perspective, the following list provides a general summary of the polygon aligned leaf BSP tree construction process:

1) The first step is to select a polygon from those passed into this function that will be used as the candidate plane for a new node. This polygon should be removed from further consideration as a node plane candidate.

2) Loop through each polygon in the list, including the one selected in the previous step.

3) Classify the current polygon against the node plane and add it to the appropriate child list depending on the result of the classification.

4) If the polygon was found to span the node plane, split the polygon against this plane in order to generate two new polygon fragments. Add each of these fragments to the appropriate child list and release the original polygon.

5) Once each polygon has been processed, determine if there are any remaining candidates in front or behind and attach a new leaf or node depending on the outcome of that test.

6) If new nodes were generated, recurse down into those nodes and begin the process again.

As you can see the general construction process remains the same in principle as that employed by the node BSP tree, and even the kD-tree in part. There are, however, a few intricacies and additional steps not included in this list that we will discover as we take a look at how this function is implemented.

```
HRESULT CBSPTree::BuildBSPTree( unsigned long Node, CBSPFace * pFaceList )
{
    // 49 Bytes including Parameter list (based on __thiscall declaration)
    CBSPFace     *TestFace   = NULL, *NextFace  = NULL;
    CBSPFace     *FrontList  = NULL, *BackList  = NULL;
    CBSPFace     *FrontSplit = NULL, *BackSplit = NULL;
    CBSPFace     *Splitter   = NULL;
    HRESULT       ErrCode    = BC_OK;
    CLASSIFYTYPE  Result;
    int           v;

    // Log – BSP: Update Progress
```

This function accepts two parameters. The first is the index to the current node being generated at this point in the recursive process. When calling the function for the first time, this parameter will contain a value of 0 signifying the root node. The second parameter is the first (or head) item of the linked list containing each of the polygons to be processed and classified at this level in the tree.

With this information to hand, the first task that this function must undertake is to select an appropriate node plane candidate – or splitter polygon – from the list passed to this level of the recursive build procedure. This is achieved with a call to the ‘SelectBestSplitter’ method that we first encountered in lab project 16.1. To this function we pass the current list of available polygons, the number of polygons that we would like to sample for selection as well as the split heuristic constant. The ‘SelectBestSplitter’ function will search through the list specified and select an appropriate polygon from those that has not already been used as a splitter in the past. If everything was successful, a reference to this selected

polygon will be returned from the selection routine and stored in the local 'Splitter' variable for use throughout the remainder of this function.

Now that we have an appropriate polygon that can be used as the candidate plane for the node at this level, this polygon's 'UsedAsSplitter' member is set to true. This is done in order to remove it from consideration during further calls to the 'SelectBestSplitter' function. The index stored in the 'Plane' member of this polygon is also then copied into the match member within the current node.

```
    // Select the best splitter from the list of faces passed
    Splitter = SelectBestSplitter( pFaceList, m_OptionSet.SplitterSample,
                                   m_OptionSet.SplitHeuristic);
    if (!Splitter) { ErrCode = BCERR_BSP_INVALIDGEOMETRY; goto BuildError; }

    // Flag as used, and store plane
    Splitter->UsedAsSplitter = true;
    GetNode(Node)->Plane     = Splitter->Plane;
```

Once we have selected the candidate splitter polygon used to generate the node at this stage we then begin the process of iterating through the list of polygons passed in to this level of the recursive procedure in the same way as we have done several times in the past.

```
    // Begin face iteration....
    for ( TestFace = pFaceList; TestFace != NULL;
          TestFace = NextFace, pFaceList = NextFace )
    {
        // Store plane for easy access
        CPlane3 * pPlane = GetPlane( TestFace->Plane );

        // Store next face, as 'TestFace' may be modified / deleted
        NextFace = TestFace->Next;
```

One of the most beneficial aspects of storing indices to the shared plane information within each polygon is that we can determine whether two faces or nodes are coplanar with one another simply by testing for equality between their plane indices. As you might imagine, this could potentially save us from having to perform hundreds or even thousands of plane distance calculations that might be required when testing each vertex via the 'ClassifyPoly' method. Of course, if the plane indices do not match then we must still perform the more traditional polygon classification step, but the savings gained make this step more than worthwhile. From an accuracy standpoint the benefits gained by performing this simple check are also significant because we do not need to use the more inaccurate vertex classification process to determine those polygons that are coplanar with the separating plane.

In the following code you can see this technique employed. If the polygon we are testing has a plane index that is equal to that of the splitter chosen earlier, then we simply store a value of 'CLASSIFY_ONPLANE' in the result variable. If the plane indices do not match however, then we call the plane's 'ClassifyPoly' method to perform the more traditional classification step.

```
        // Classify the polygon
        if ( TestFace->Plane == Splitter->Plane )
        {
```

```
            Result = CLASSIFY_ONPLANE;

        } // End if uses same plane
        else
        {
            Result = GetPlane(Splitter->Plane)->ClassifyPoly(TestFace->Vertices,
                                                             TestFace->VertexCount,
                                                             sizeof(CVertex));

        } // End if differing plane
```

The co-planar case is arguably the most critical of each of the classification cases which is one of the reasons why the aforementioned plane index test is so beneficial. In this on-plane case we must first determine if the polygon faces in the same direction as the current node plane. In our implementation we can use the plane's 'SameFacing' method to test for this.

If the polygon is found to face in the same direction as this coplanar node then we can automatically remove that polygon from being considered as a node plane candidate in the future. If it was not marked as used at this point, then any node generated from it would simply describe the same separating plane as that of the current node. Once this is done, we must then add the current polygon to the linked list representing those polygons or polygon fragments contained in the front halfspace of the node – the linked list headed by the local 'FrontList' variable.

If the polygon does not point in the same direction as the plane then, as we know, this polygon should *not* be removed from consideration in the leaf based BSP tree, and must go on to generate its own node. This polygon must be added to the linked list representing those polygons or polygon fragments contained in the back halfspace of the node – the linked list headed by the local 'BackList' variable.

```
        // Classify the polygon against the selected plane
        switch ( Result )
        {
            case CLASSIFY_ONPLANE:

                // Test the direction of the face against the plane.
                if ( GetPlane(Splitter->Plane)->SameFacing( pPlane->Normal ) )
                {
                    // Mark matching planes as used
                    if (!TestFace->UsedAsSplitter)
                    {
                        TestFace->UsedAsSplitter = true;

                        // Log – BSP : Update Progress Details

                    } // End if !UsedAsSplitter

                    TestFace->Next  = FrontList;
                    FrontList       = TestFace;
                }
                else
                {
                    TestFace->Next  = BackList;
                    BackList        = TestFace;
```

```
            } // End if Plane Facing
            break;
```

The front classification case is relatively straight forward. If we step into this case it means that the polygon is contained completely within the front halfspace of the node. As a result, this polygon is attached directly to the linked list referenced by the 'FrontList' variable.

```
        case CLASSIFY_INFRONT:
            // Pass the face straight down the front list.
            TestFace->Next      = FrontList;
            FrontList           = TestFace;
            break;
```

In a similar fashion to the front case, if the polygon we are testing is found to be completely contained within the back halfspace of the node's plane then it is simply attached to the linked list referenced by the local 'BackList' variable.

```
        case CLASSIFY_BEHIND:
            // Pass the face straight down the back list.
            TestFace->Next      = BackList;
            BackList            = TestFace;
            break;
```

If we step into the spanning case, we have of course found a situation in which the polygon intersects the current node plane, with a portion of that polygon found to lie on either side. This being the case, we must split this polygon into those two fragments separated by the plane in order to pass the relevant portion of the polygon down either side of the node. Before we can do this however we must first allocate the two new polygon objects that will contain these split fragments. To make the handling of errors a little easier in this application, we have implemented a function named 'AllocBSPFace' – covered shortly – that can be used to retrieve a new 'CBSPFace' instance. This is called twice, storing the two resulting polygon pointers into the 'FrontSplit' and 'BackSplit' variables respectively.

```
        case CLASSIFY_SPANNING:
            // Ensure this is not an invalid operation
            // Allocate new front fragment
            if (!(FrontSplit = AllocBSPFace()))
            {
                ErrCode = BCERR_OUTOFMEMORY;
                goto BuildError;
            }
            FrontSplit->Next    = FrontList;
            FrontList           = FrontSplit;

            // Allocate new back fragment
            if (!(BackSplit     = AllocBSPFace()))
            {
                ErrCode = BCERR_OUTOFMEMORY;
                goto BuildError;
            }
            BackSplit->Next     = BackList;
            BackList            = BackSplit;
```

Notice in the above code that we have immediately added the new polygon objects directly to the front and back lists. We do this in order to ensure that these fragments are automatically released – along with the others in each list – should an error occur in the following split operation.

Having allocated these two new polygon objects, we can now call the spanning polygon's 'Split' method, passing in the separating plane along with the two new polygon objects to be generated. Should it be successful, this will result in the 'FrontSplit' and 'BackSplit' polygons describing the portions of the original polygon that lay on either side of the plane.

```
                // Split the polygon
                if (FAILED( ErrCode = TestFace->Split(*GetPlane(Splitter->Plane),
                                                      FrontSplit,
                                                      BackSplit))) goto BuildError;
```

Recall in the earlier coverage of the 'BuildPlaneArray' method, we described a mechanism in which the vertices of each polygon were pushed back such that they always touched the surface of the plane they reference. Because the polygon splitting operation is prone to floating point accuracy errors – like most other operations – we perform this operation once again here on both the new front and back polygon fragments. While this is not a critical step in the generation of a polygon-aligned BSP tree, this step may help to greatly improve the accuracy of the compilation process.

```
                // Ensure that all vertices are on the original plane
                for ( v = 0; v < (signed)FrontSplit->VertexCount; v++ )
                {
                    float result = FrontSplit->Vertices[v].Dot( pPlane->Normal ) +
                                   pPlane->Distance;

                    FrontSplit->Vertices[v] += (pPlane->Normal * -result);

                } // Next Vertex

                // Ensure that all vertices are on the original plane
                for ( v = 0; v < (signed)BackSplit->VertexCount; v++ )
                {
                    float result = BackSplit->Vertices[v].Dot( pPlane->Normal ) +
                                   pPlane->Distance;

                    BackSplit->Vertices[v] += (pPlane->Normal * -result);

                } // Next Vertex
```

At this stage, we have generated the two new polygon fragments that describe those portions of the original polygon that lay on either side of the plane. We have also added them to the front and back lists ready to be processed at the next level in the recursive procedure. As a result, we have no further need for the original polygon from which these two fragments were generated and therefore we simply release it here. Having removed this original polygon, and inserted two additional polygons into the compile process, the number of active faces will have increased by one. To this end we increment the 'm_lActiveFaces' member here to ensure that our progress is accurately reported before moving on to the next polygon in the linked list passed at this level in the recursive process.

```
                // Log – BSP: Update Progress Details

                // + 2 Fragments - 1 Original
                m_lActiveFaces++;

                // Free up original face
                delete TestFace;

                break;

        } // End Switch

    } // End while loop
```

With every polygon now having been processed, and the resulting front and back lists populated we can now begin to process this information.

During the earlier coverage of the settings contained within the ‘BSPOPTIONS’ structure we discussed a technique that could be employed to help ensure the integrity of the solid/empty leaf information within the tree. As mentioned, the ‘RemoveBackLeaves’ flag contained within the ‘m_OptionSet’ member allows the application to specify whether or not it would like the BSP compiler to enforce the removal of any polygon fragments which end up behind a node where solid space should exist.

If this process was enabled then we first test the contents of the back list to determine whether or not any further nodes will be generated behind the current one. This achieved with a call to the ‘CountSplitters’ method that is responsible for totalling those polygons in the specified linked list that have not yet been used as node plane candidates. If this function returns a value equal to 0 after having been passed the back list, then we know that we have reached a node behind which solid space exists. Given that this is the case, the following code then proceeds to delete each of the polygons contained within the back list that would have ended up in this solid space.

```
    // Should We Back Leaf Cull ?
    if ( m_OptionSet.RemoveBackLeaves )
    {
        // If No potential splitters remain, free the back list
        if ( BackList && CountSplitters( BackList ) == 0 )
        {
            // Release illegal polygon fragments
            for ( TestFace = BackList; TestFace; TestFace = TestFace->Next )
            {
                delete TestFace;
                m_lActiveFaces--;

            } // End if
            BackList = NULL;

        } // End if No Splitters

    } // End If RemoveBackLeaves
```

At this stage in the process we have access to the final front and back polygon linked lists. This is the ideal time for us to generate the final bounding box extents for the current node based on this information. As a result, we make two calls to the current node's 'CalculateBounds' method. In the first call we pass the front polygon list to the first parameter, and specify that the node's bounding box should be reset by passing a value of 'true' to the second. Once this function returns the node's 'Bounds' members will contain extents values which describe those polygons in the front list. We must take into account *every* polygon that will exist at some point below the node however. As a result we call this method again, passing in the back polygon list and a value of false to the second parameter which instructs the 'CalculateBounds' function that it should not reset the extents. Instead it will grow the current extents to include those polygons passed.

```
    // Calculate the nodes bounding box
    GetNode(Node)->CalculateBounds( FrontList, true );
    GetNode(Node)->CalculateBounds( BackList, false );
```

When processing the front of the node, we know that there can never be a situation in which no polygon data exists within the linked list referenced by the 'FrontList' variable. This is due to the fact that the polygon used as the node plane candidate will always at least be added to the front list. In the following code block therefore we begin by counting the number of splitters that remain in that list.

Recall that a leaf is normally created at the point when there are no further nodes that can be generated from those polygons contained in the appropriate list – e.g. each polygon has its 'UsedAsSplitter' member set to true or the list contains no polygon data at all. Since the latter of these two cases is not possible in front of the node, this means that there must be at least one polygon stored here but it has already been used.

The first case in the conditional if/else statement in the following code block creates a new leaf structure should there have been no available splitter polygons contained in the front list. This leaf is attached to the current node's 'Front' member in the manner outlined in the 'File Based Data Structures' section in which we discussed how leaves and nodes are referenced in this application. After the leaf has been attached to the node in this way, we then make a call to the 'ProcessLeafFaces' method. To this call we pass the newly generated leaf structure and the front polygon list in order to have this function insert those polygons into the leaf structure on our behalf.

```
    // Is the front list empty?
    if ( CountSplitters( FrontList ) == 0 )
    {
        // Add a new leaf and store the resulting faces
        if (!IncreaseLeafCount())
        {
            ErrCode = BCERR_OUTOFMEMORY;
            goto BuildError;
        }

        GetNode(Node)->Front = -((long)GetLeafCount());
        if (FAILED(ErrCode = ProcessLeafFaces( GetLeaf( GetLeafCount() - 1 ),
                                               FrontList ))) goto BuildError;
    }
```

The second case in this conditional if/else statement is executed if there were polygons in the front list that are still available for use as node plane candidates – e.g. at least one of these polygons has a 'UsedAsSplitter' member set to 'false'. In this case there are obviously further nodes that can be generated in front of this node. As a result, we generate a new node structure, store the index to that new node in the 'Front' member of the current node and finally step down the front of this node by recursively calling the 'BuildBSPTree' function here.

```
    else
    {
        // Allocate a new node and step into it
        if (!IncreaseNodeCount())
        {
            ErrCode = BCERR_OUTOFMEMORY;
            goto BuildError;
        }

        GetNode(Node)->Front = GetNodeCount() - 1;
        ErrCode = BuildBSPTree( GetNode(Node)->Front, FrontList );

    } // End If FrontList

    // Front list has been passed off, we no longer own these
    FrontList = NULL;
    if ( FAILED(ErrCode) ) goto BuildError;
```

If you have read the textbook at this point you should be aware that whenever the list of available polygons that fall *behind* a particular node are depleted, then the space behind this node will describe solid space. Recall that a leaf is normally created at the point when there are no further nodes that can be generated from those polygons contained in the appropriate list – e.g. each polygon has its 'UsedAsSplitter' member set to true or the list contains no polygon data at all. In either case, when the back list is empty, we must attach a leaf to the back of the current node that denotes this solid area. Due to the fact that there are no polygons which remain to be collected into the leaf however, it would be wasteful if we allocated and store a new leaf structure here. Instead, we take advantage of the fact that we are using *indices* to reference any leaves attached to either side of the node in order to store a special case 'solid leaf' constant code defined as follows.

```
#define BSP_SOLID_LEAF 0x80000000
```

This value is equal to the largest *negative* value that can be stored in a signed 32 bit variable. Because this constant defines a negative value, the application will still be able to test for the existence of a leaf attached to the back of a node with a simple '<0' comparison test. In order to determine if a leaf is solid however, the 'Back' member of the node must be tested explicitly for equality with this solid leaf define.

In the following code we first check to see if the list of polygons is empty simply by testing the 'BackList' variable to see if it contains a zero or NULL value. If this is found to be the case then we know that solid space must exist behind this node and as a result we set the current node's 'Back' member value to the 'BSP_SOLID_LEAF' constant defined earlier.

```
    // Is the back list empty?
    if ( !BackList )
```

```
    {
        // Set the back as a solid leaf
        GetNode(Node)->Back = BSP_SOLID_LEAF;
    }
```

If the back list *does* contain polygon data at this stage then either the 'RemoveBackLeaves' compile option was disabled by the application, or this backlist simply contains no remaining splitters. In either case, we treat this situation in exactly the same manner as we did with the front list – e.g. create, attach and populate a new leaf structure to the back of this node if there were no splitters remaining or; create and attach a new node behind the current, and recurse into the back of the current node with a further call to the 'BuildBSPTree' function.

```
    else
    {
        // No splitters remaining?
        if ( CountSplitters( BackList ) == 0 )
        {
            // Add a new leaf and store the resulting faces
            if (!IncreaseLeafCount())
            {
                ErrCode = BCERR_OUTOFMEMORY;
                goto BuildError;
            }

            GetNode(Node)->Back = -((long)GetLeafCount());
            if (FAILED(ErrCode = ProcessLeafFaces( GetLeaf( GetLeafCount() - 1 ),
                                                   BackList ))) goto BuildError;

        } // End if no splitters
        else
        {
            // Allocate a new node and step into it
            if (!IncreaseNodeCount())
            {
                ErrCode = BCERR_OUTOFMEMORY;
                goto BuildError;
            }

            GetNode(Node)->Back = GetNodeCount() - 1;
            ErrCode = BuildBSPTree( GetNode(Node)->Back, BackList);

        } // End if remaining splitters

    } // End If BackList

    // Back list has been passed off, we no longer own these
    BackList = NULL;
    if ( FAILED(ErrCode) ) goto BuildError;

    // Success
    return BC_OK;
```

There were several cases within the 'BuildBSPTree' function in which we jumped to the section of code referenced by the label 'BuildError' by using a goto statement. While some might argue that using a

try/catch exception handling method would be more compatible with the C++ standards specification, we have chosen this method within this recursive procedure purely for reasons of efficiency and convenience.

Regardless of how we reach this block of error handling logic, the important point is how each of the front and back lists maintained by this function are being handled in addition to the list of polygons passed into this function. In the following code you can see that each linked list is passed into the 'TrashFaceList' utility function that is designed to add each polygon in the specified list to the 'm_vpGarbage' member array in order for them to be released at a later time. We will discuss why this is the case in the coverage of the 'TrashFaceList' function later in this section.

The final part of this error handling logic returns the error information currently stored in the local 'ErrCode' variable – specified by the main portion of this code – to the calling function. Remember, that because this is a recursive procedure, the function responsible for making the call might actually be this very same one. As a result, we must remember to pay attention to any error codes returned from these recursive calls. If an error was returned from a child recursion, then that call must also begin to clean up and return any applicable error to its parent. This would therefore be repeated until we return all the way back out of each level in the call recursion.

```
BuildError:
    // Add all currently allocated faces to garbage heap
    TrashFaceList( pFaceList );
    TrashFaceList( FrontList );
    TrashFaceList( BackList  );

    // Failed
    return ErrCode;
}
```

CBSPTree::SelectBestSplitter

We encountered the 'SelectBestSplitter' function during our coverage of lab project 16.1. There are however several minor modifications that we must discuss in order to make this function compatible with the BSP leaf tree and our new pre-processing tool.

```
CBSPFace * CBSPTree::SelectBestSplitter( CBSPFace * pFaceList,
                                         unsigned long SplitterSample,
                                         float SplitHeuristic )
{
    unsigned long Score, Splits, BackFaces, FrontFaces;
    unsigned long BestScore = 10000000, SplitterCount = 0;
    CBSPFace *Splitter = pFaceList, *CurrentFace = NULL, *SelectedFace = NULL;
```

Other than using slightly different types (CBSPFace instead of CPolygon for instance) the parameters, local variable list and return value remain similar to that of our previous implementation of this function.

One thing that has changed however is the data that will be passed in to this function. In the previous implementation of the BSP node tree class, the 'SelectBestSplitter' function would be passed a list of only those polygons that had not been linked to nodes in the tree and removed from further processing.

In the BSP leaf tree information, remember that the polygon data is now being passed through the tree in order for it to be collected within the leaves. As a result, this function will now receive even those polygons already used as node plane candidates. For this reason, in the following while loop, this function must ignore any polygon that has its 'UsedAsSplitter' member set to 'true'.

```
    // Traverse the Face Linked List
    while ( Splitter != NULL )
    {

        // If this has NOT been used as a splitter then
        if ( !Splitter->UsedAsSplitter )
        {
            // Create testing splitter plane
            CPlane3 SplittersPlane( Splitter->Normal, Splitter->Vertices[0] );
```

The only remaining difference of any significance within this function is the way in which the splitter plane is generated and tested. In our previous implementation we made use of the D3DXPLANE structure in conjunction with our collision library's 'PolygonClassifyPlane' function. In this lab project however, we make use of the new 'CPlane3' math utility class and its associated 'ClassifyPoly' routine to do the same job.

Other than these minor differences, the mechanism by which a polygon is selected as a potential splitter remains the same. As a result the remainder of this function is included only in the interest of completeness.

```
            // Test against the other poly's and count the score
            CurrentFace = pFaceList;
            Score = Splits = BackFaces = FrontFaces = 0;
            while ( CurrentFace != NULL )
            {
                CLASSIFYTYPE Result =
                         SplittersPlane.ClassifyPoly(CurrentFace->Vertices,
                                                     CurrentFace->VertexCount,
                                                     sizeof(CVertex) );
                switch ( Result )
                {
                    case CLASSIFY_INFRONT:
                        FrontFaces++;
                        break;

                    case CLASSIFY_BEHIND:
                        BackFaces++;
                        break;

                    case CLASSIFY_SPANNING:
                        Splits++;
                        break;

                    default:
                        break;

                } // switch
```

```
                CurrentFace = CurrentFace->Next;

            } // Next Face

            // Tally the score (modify the splits * n )
            Score = (unsigned long)((long)abs( (long)(FrontFaces - BackFaces) ) +
                                                    (Splits * SplitHeuristic));
            // Is this the best score ?
            if ( Score < BestScore)
            {
                BestScore    = Score;
                SelectedFace = Splitter;
            } // End if better score

            SplitterCount++;

        } // End if this splitter has not been used yet

        Splitter = Splitter->Next;

        // Break if we reached our splitter sample limit.
        if (SplitterSample != 0 && SplitterCount >= SplitterSample && SelectedFace)
            break;

    } // Next Splitter

    // This will be NULL if no faces remained to be used
    return SelectedFace;
}
```

CBSPTree::CountSplitters

This method serves as utility function utilized by the 'BuildBSPTree' function to quickly count the number of polygons that have not yet been selected for node plane candidacy within the specified 'CBSPFace'object linked list.

Using the same linked list iteration logic we have employed several times before in this class, the 'CountSplitters' function simply tests the value of each polygon's Boolean 'UsedAsSplitter' member. If this member contains a value of 'false' this indicates that the polygon has not yet been used in the generation of a node. Should this be the case, the counter describing the total number of *unused* polygons is then incremented. This value is maintained within the local 'SplitterCount' variable declared at the top of this function and initialized with a default value of 0. Once each polygon within the linked list has been tested, this function then returns to the calling function passing back this total number of unused polygons.

```
unsigned long CBSPTree::CountSplitters( CBSPFace * pFaceList ) const
{
    unsigned long SplitterCount = 0;
    CBSPFace     *Iterator      = pFaceList;

    // Count the number of splitters
    while ( Iterator != NULL )
```

```
    {
        if ( !Iterator->UsedAsSplitter ) SplitterCount++;
        Iterator = Iterator->Next;

    } // End If

    // Return number of splitters remaining
    return SplitterCount;
}
```

CBSPTree::ProcessLeafFaces

As we already know at this point, the 'ProcessLeafFaces' method is called by the 'BuildBSPTree' function in order to have the polygons contained within the linked list (passed into the second parameter) added to the specified leaf (passed into the first). However, what may not initially have been clear is the reason behind why this was necessary.

As we know, there are two types of results which we can obtain from the BSP tree compiler. Either the leaves reference those polygons that may have been split during the compilation process, or they reference the original unsplit polygons stored in the tree's final polygon array at the start of the process.

This function wraps the logic required for both of these types of tree result. The first of these cases is the non-split resulting tree type. If this type of tree was selected by the application then the polygon linked list passed to this function is sent straight through to the leaf's 'BuildFaceIndices' function. Recall that the polygon list sent to us from the 'BuildBSPTree' function contains those polygons that have been split against and passed through the tree during the compilation process. These polygon fragments each store a value in their 'OriginalIndex' member that references the original unsplit polygon already stored in the tree's polygon array. As a result, the indices for the original unsplit polygons are added directly to the leaf. Because of the fact that we are only interested in the original unsplit versions of those polygons passed to this function in the non-split case, the polygons contained in this list are released with a call to the 'FreeFaceList' utility method.

```
HRESULT CBSPTree::ProcessLeafFaces( CBSPLeaf * pLeaf, CBSPFace * pFaceList )
{
    HRESULT ErrCode = BC_OK;

    // Depending on the tree type we may need to store these resulting faces
    if ( m_OptionSet.TreeType == BSP_TYPE_NONSPLIT )
    {
        // Add these faces to the leaf
        if ( FAILED( ErrCode = pLeaf->BuildFaceIndices( pFaceList )))
             return ErrCode;

        // Release the faces
        FreeFaceList( pFaceList );

    } // End if NSR Type
```

The alternative case is the standard split resulting tree in which the polygon data that has been passed through the tree – and potentially split into many additional polygons – is to be stored in the tree's final

polygon array. As we loop through the list of polygons passed to this function in the split treetype case, we simply allocate a new element in the 'm_vpFaces' array with a call to the 'IncreaseFaceCount' method. Into this element we place the actual pointer to the current polygon object that has been used in the tree compilation process. Finally we update its 'OriginalIndex' member with its own position in the tree's polygon list before passing each of these polygons into the leaf's 'BuildFaceIndices' function as before.

```
    else
    {
        // Split tree, we need to store the face pointers
        CBSPFace * Iterator = pFaceList;
        while ( Iterator != NULL )
        {
            if (!IncreaseFaceCount()) return BCERR_OUTOFMEMORY;
            SetFace( GetFaceCount() - 1, Iterator );

            // Set our original index to the final position it
            // is stored in the array (Split type only)
            Iterator->OriginalIndex = GetFaceCount() - 1;
            Iterator = Iterator->Next;

        } // Next Face

        // Add these faces to the leaf
        if ( FAILED( ErrCode = pLeaf->BuildFaceIndices( pFaceList )))
                    return ErrCode;

    } // End if Split Type

    // Success
    return BC_OK;
}
```

CBSPTree::IncreaseNodeCount / IncreaseLeafCount / IncreasePlaneCount

There are four main utility functions defined within the 'CBSPTree' class that are responsible for resizing the various STL vector members maintained by each instance of this class. The three functions outlined here are each responsible for increasing the size of the node, leaf and plane arrays by one element in each call. The fourth function of this type is responsible for resizing the polygon array but this is implemented in a slightly different manner and is covered independently.

The first of these is the 'IncreaseNodeCount' method. This function is used to allocate and insert a new node object at the end of the BSP tree's internal node array. Due to the fact that this function may be called many times to add a large number of nodes during the BSP compilation process, we use the standard capacity threshold resizing logic in order to prevent this array from being resized / reallocated for every node we add. Put simply, if the total number of elements in the vector *including* the new node is found to exceed its current internal capacity, then we will proceed to allocate additional space by calling the vector's 'Reserve' method. To this method we specify a capacity equal to the current number of items stored in the vector plus an additional number of slack elements into which the vector can grow.

In this way, the vector will only ever be physically resized each time we have added a number of nodes equal to this additional slack space. The number of additional elements reserved in this function is defined by the constant shown below.

```
#define BSP_ARRAY_THRESHOLD 100
```

Once we have reserved any required number of elements in the 'm_vpNodes' member, we can then allocate the new node, and add it to this member with a call to the STL vector's 'push_back' method.

```
bool CBSPTree::IncreaseNodeCount()
{
    CBSPNode *NewNode = NULL;

    try
    {
        // Resize the vector if we need to
        if (m_vpNodes.size() >= (m_vpNodes.capacity() - 1))
        {
            m_vpNodes.reserve( m_vpNodes.size() + BSP_ARRAY_THRESHOLD );
        } // End If

        // Allocate a new Node ready for storage
        // Note : VC++ new does not throw an exception on failure (easily ;)
        if (!(NewNode = new CBSPNode)) throw std::bad_alloc();

        // Push back this new Node
        m_vpNodes.push_back(NewNode);
    } // Try vector ops
```

Each of these functions also implements exception handling logic similar to that shown below. This prevents the BSP compiler from having to wrap each insert operation with its own exception handling logic because any exceptions will have been handled already.

```
    // Catch Failures
    catch (std::bad_alloc)
    {
        return false;
    }
    catch (...)
    {
        if (NewNode) delete NewNode;
        return false;
    } // End Catch

    // Success
    return true;
}
```

We won't go into detail about the remaining two 'Increase*' methods as their functionality is essentially identical. In the interest of completeness however the code to the 'IncreaseLeafCount' method is shown below. This function is used to allocate and insert a new leaf object at the end of the BSP tree's internal leaf array.

```
bool CBSPTree::IncreaseLeafCount()
{
    CBSPLeaf *NewLeaf = NULL;

    try
    {
        // Resize the vector if we need to
        if (m_vpLeaves.size() >= (m_vpLeaves.capacity() - 1))
        {
            m_vpLeaves.reserve( m_vpLeaves.size() + BSP_ARRAY_THRESHOLD );
        } // End If

        // Allocate a new leaf ready for storage
        // Note : VC++ new does not throw an exception on failure (easily ;)
        if (!(NewLeaf = new CBSPLeaf)) throw std::bad_alloc();

        // Push back this new leaf
        m_vpLeaves.push_back(NewLeaf);
    } // Try vector ops

    // Catch Failures
    catch (std::bad_alloc)
    {
        return false;
    }
    catch (...)
    {
        if (NewLeaf) delete NewLeaf;
        return false;
    } // End Catch

    // Success
    return true;
}
```

Finally, the BSP tree class also defines the 'IncreasePlaneCount' method. This function is used to allocate and insert a new plane object at the end of the BSP tree's combined plane array.

```
bool CBSPTree::IncreasePlaneCount()
{
    CPlane3 *NewPlane = NULL;

    try
    {
        // Resize the vector if we need to
        if (m_vpPlanes.size() >= (m_vpPlanes.capacity() - 1))
        {
            m_vpPlanes.reserve( m_vpPlanes.size() + BSP_ARRAY_THRESHOLD );
        } // End If

        // Allocate a new plane ready for storage
        // Note : VC++ new does not throw an exception on failure (easily ;)
        if (!(NewPlane = new CPlane3)) throw std::bad_alloc();
```

```
        // Push back this new plane
        m_vpPlanes.push_back(NewPlane);
    } // Try vector ops

    // Catch Failures
    catch (std::bad_alloc)
    {
        return false;
    }
    catch (...)
    {
        if (NewPlane) delete NewPlane;
        return false;
    } // End Catch

    // Success
    return true;
}
```

CBSPTree::IncreaseFaceCount

As with each of the memory management functions covered so far, this function increases the capacity of the BSP trees internal polygon array using a similar threshold mechanism. There is one major difference with this function when compared to those discussed previously. In this function we do *not* allocate and insert a new 'CBSPFace' object. Instead we simply push a NULL pointer value onto the end of the 'm_vpFaces' STL vector. This behavior was selected due to the fact that in the majority of cases, any polygon data that will be stored within the 'm_vpFaces' array has already been allocated prior to the compilation step. As a result, any existing polygon objects would simply need to be transferred into this array with a call to the 'SetFace' accessor function.

```
bool CBSPTree::IncreaseFaceCount()
{

    try
    {
        // Resize the vector if we need to
        if (m_vpFaces.size() >= (m_vpFaces.capacity() - 1))
        {
            m_vpFaces.reserve( m_vpFaces.size() + BSP_ARRAY_THRESHOLD );

        } // End If

        // Push back a NULL pointer
        m_vpFaces.push_back( NULL );

    } // Try vector ops

    // Catch Failures
    catch (...)
    {
        return false;

    } // End Catch
```

```
    // Success
    return true;
}
```

CBSPTree::AllocBSPFace

The ‘AllocBSPFace’ method is yet another utility function designed to make the job of implementing the more comprehensive BSP leaf tree compilation process a little less clumsy. This function is designed simply to allocate a single ‘CBSPFace’ object that will be passed back via the return value. This could of course be achieved simply by allocating the polygon object within the calling function using the ‘new’ operator. However, rather than have the compiler wrap each allocation attempt in exception handling logic, this function does that on its behalf.

The single parameter accepted by this function is used to specify any ‘CFace' object that should be duplicated into the new ‘CBSPFace’ instance using the overloaded constructor of that class.

```
CBSPFace * CBSPTree::AllocBSPFace( const CFace * pDuplicate )
{
    CBSPFace * NewFace = NULL;

    try
    {
        // Call the corresponding constructor
        if (pDuplicate != NULL)
        {
            NewFace = new CBSPFace( pDuplicate );
        }
        else
        {
            NewFace = new CBSPFace;
        } // End If pDuplicate

        // Note : VC++ new may not throw an exception on failure (easily ;)
        if (!NewFace) throw std::bad_alloc();

    } // End Try

    catch (...) { return NULL; }

    // Success!
    return NewFace;
}
```

CBSPTree::FreeFaceList

The ‘FreeFaceList’ method is a utility function designed to **immediately** release any polygon data stored within the linked list of ‘CBSPFace’ object pointers passed to its single parameter. We observed this method being used during the earlier coverage of the ‘ProcessLeafFaces’ function.

In the following code we can see how this function simply iterates through each element in the specified linked list using the traversal logic we have seen throughout this application. For each polygon encountered in this loop we must first make a duplicate of the pointer stored within that polygon's 'Next' member used to iterate to the next polygon in the list. We must take this action first due to the fact that the current iterator polygon will be deleted in the very next step, rendering that polygon and its members invalid. Before moving on to the next polygon using this duplicated 'NextFace' pointer, we decrement the BSP tree's 'm_lActiveFaces' member due to there now being one less active polygon within the BSP compilation process.

Once the loop completes, this function simply returns control back to the calling function having released every polygon in the list passed.

```
void CBSPTree::FreeFaceList( CBSPFace * pFaceList )
{
    // Free up the linked list
    CBSPFace * TestFace = pFaceList, *NextFace = NULL;
    while ( TestFace != NULL )
    {
        NextFace = TestFace->Next;
        delete TestFace;
        m_lActiveFaces--;
        TestFace = NextFace;
    } // Next Face
}
```

CBSPTree::TrashFaceList

Whenever an error occurs during the construction of the tree, there are most likely many references to the original polygons and new polygons alike that exist in one or more of the lists contained on the stack at each level in the recursion. Under failure conditions, it will be difficult to safely release any allocated polygon data as the recursion unwinds due to the fact that we may attempt to release the same polygon instance on more than one occasion. This would of course cause an illegal memory access and a whole new problem to deal with.

To ensure that we can safely release each of the polygon objects referenced or created by the compile process up to the point of failure, any remaining 'CBSPFace' object pointers can be inserted into the garbage collection member array by passing them to this function. As the code iterates through each of the polygons passed into this procedure, we first check to see if a pointer to that particular polygon already exists within the 'm_vpGarbage' STL vector. Only if no duplicate pointer is found to exist will this function add that polygon to the list. Once the recursive compile process has returned all the way back out to the original calling function, each of the polygons added to the garbage collection list by this function can then safely be released.

```
void CBSPTree::TrashFaceList( CBSPFace * pFaceList )
{
    CBSPFace * Iterator = pFaceList;

    while ( Iterator != NULL )
    {
```

```
        // If it doesn't already exist then add it to the list
        if (std::find(m_vpGarbage.begin(), m_vpGarbage.end(),
            Iterator) == m_vpGarbage.end())
        {

            try
            {
                // Resize the vector if we need to
                if (m_vpGarbage.size() >= (m_vpGarbage.capacity() - 1))
                {
                    m_vpGarbage.reserve( m_vpGarbage.size() +
                                         BSP_ARRAY_THRESHOLD );
                } // End If

                // Push back this new item
                m_vpGarbage.push_back(Iterator);

            } // End Try

            // On exception, we can do nothing but bail and leak
            catch (std::exception&) { return; }

        } // End If Exists

        Iterator = Iterator->Next;

    } // Next Face
}
```

The CProcessHSR Module Class

With the BSP compilation topics out of the way, we can now move onto our coverage of the hidden surface removal processing class that is actually executed in advance of the BSP compile process. It was important that we first discussed the core functionality within the BSP compiler class before moving on to this topic because the HSR process relies so heavily on the information generated by the BSP tree compilation process.

```
class CProcessHSR
{
public:
    // Constructors & Destructors for This Class.
             CProcessHSR();
    virtual ~CProcessHSR();

    // Public Member Functions Omitted

private:
    // Private Variables for This Class.
    vectorMesh      m_vpMeshList;       // List of meshes for HSR Processing.
    CMesh          *m_pResultMesh;      // The resulting unioned mesh

    HSROPTIONS      m_OptionSet;        // HSR Options Set
    ILogger        *m_pLogger;          // Logging output interface
    CCompiler      *m_pParent;          // Parent Compiler Pointer
```

```
};
```

Looking at the declaration for the HSR module class, we can see that this one-off processing module is relatively simple when compared to the BSP compiler class / module covered earlier. This class declaration is a more typical example of the kind of one-time processing module classes that we might develop within this application. There are only a few member variables declared here, so let us cover them very briefly.

vectorMesh m_vpMeshList
This member maintains a list of references to each mesh to be merged together within the hidden surface removal process. It should be populated by the application using the 'AddMesh' method defined by this class. This member variable is of the type 'vectorMesh' which is a typedef of a standard STL vector template designed to store a series of 'CMesh' object pointers.

```
typedef std::vector<CMesh*>    vectorMesh
```

CMesh * m_pResultMesh
Once the hidden surface removal process has run to its conclusion, this member variable will store a pointer to the new mesh that resulted from the union operation. This mesh object can subsequently be retrieved by the compiler class with a call to the 'GetResultMesh' method covered shortly.

HSROPTIONS m_OptionSet
This member stores the various settings, specified by the application to inform the compiler how the hidden surface removal operation should be executed. Although this structure contains no settings outside of the 'Enabled' Boolean, this options structure should be set using the 'SetOptions' accessor function in case of future enhancement.

ILogger * m_pLogger
In order for the HSR processor to report progress, error and useful status information to the user, this member variable stores a pointer to the application defined logging class. While the logging feature remains optional, this member can be set with a call to the 'SetLogger' accessor function defined by this class. In this lab project, the logging class instance is passed to this object by the 'CCompiler::PerformHSR' function.

CCompiler * m_pParent
Each compilation module class maintains a member that stores a reference to the parent compiler class that created it. This member is set with a call to the 'SetParent' accessor function defined by each of these classes. This information is used by each of these processing modules in order to test the current state of the compiler to determine if the compile operation has been either paused or cancelled. Although we have omitted much of the logging and progress functionality from the code listings in this workbook, you should take a look at the source code for lab project 16.2 to see this process in action.

HSR Module Accessor Functions

There are only a few accessor functions defined within the HSR processing module class. In each case, the body for these functions can be found in the class declaration contained in the header file, rather than in the source module / .CPP file. We will not go into detail about how each of these functions is implemented, for this you should take a look at the 'CProcessHSR.h' header file contained in the 'Compiler Source' project directory.

The only accessor functions defined by the 'CProcessHSR' class are those that set the various pieces of information required by each compilation process module. These include the process options, a pointer to the parent compiler and a pointer to the class defined logging class. These functions simply store those specified values into the applicable member variables declared by this class.

```
void            SetOptions    ( HSROPTIONS Options    )
void            SetParent     ( CCompiler * pCompiler )
void            SetLogger     ( ILogger * pLogger     )
```

Now that we are familiar with each of this class's member variables, let us move on to examine the main process functionality.

CProcessHSR::AddMesh

If we refer back to the discussion of the 'CCompiler::ProcessHSR' function, we saw how the 'AddMesh' function was called in order to add each of the scene's non-detail meshes to this hidden surface removal module. This method accepts a single parameter into which is passed a single pointer to a 'CMesh' object. This mesh object pointer will simply be added to the end of the 'm_vpMeshList' member vector using the same capacity threshold mechanism that is used within several other functions in this application. Any meshes that are to be considered by this processing module must be specified in advance, prior to calling this object's 'Process' function.

```
bool CProcessHSR::AddMesh( CMesh * pMesh )
{
    try
    {
        // Resize the vector if we need to
        if (m_vpMeshList.size() >= (m_vpMeshList.capacity() - 1))
        {
            m_vpMeshList.reserve( m_vpMeshList.size() + HSR_ARRAY_THRESHOLD );

        } // End If

        // Finally add this mesh pointer to the list
        m_vpMeshList.push_back( pMesh );

    } // End Try Block

    catch (...)
    {
        // Clean up and bail
        m_vpMeshList.clear();
        return false;

    } // End Catch
```

```
    // Success
    return true;
}
```

CProcessHSR::Process

After having added each of the non-detail scene meshes to this module's source mesh array, this function is called in order to begin the actual hidden surface removal process. Before we move onto examine the implementation for this class, let us once again examine the procedure from a high level overview.

The steps involved in performing the hidden surface removal process are as follows:

1) Allocate and compile a BSP tree object for each individual mesh being processed by this module.

2) Step through each mesh (A) within an outer loop.

3) Step through each mesh (B) within an inner loop.

4) Inside the inner loop, skip any meshes that have already been processed. If the current mesh 'B' has not yet been processed, then test for intersection between mesh 'A' and mesh 'B' using a simple broad phase AABB test. If no intersection occurred, these meshes do not need to be unioned.

5) If the bounding boxes of both meshes intersected, perform an accurate intersection test between these two meshes using the BSP tree information.

6) If mesh 'A' and mesh 'B' were found to intersect with one another then first clip mesh 'A' against mesh 'B's tree, removing any parts of that mesh that exist in solid space. Then do the same in reverse such that mesh 'B' is clipped against mesh 'A's tree.

7) Repair any unnecessary splits that occurred during the clipping process for each mesh, and then exit the inner loop.

8) As the last step of the outer loop, add the surviving polygons from mesh 'A' to the resulting mesh being constructed in this process.

9) Move on to the next mesh in the outer loop and return to step 3 until all meshes have been processed in the outer loop.

Now that we are armed with a general idea of how this process should function, let us take a look at how this has process has actually been implemented in this lab project.

```
HRESULT CProcessHSR::Process()
{
    HRESULT     hRet;
```

```
    BSPOPTIONS  Options;
    int         i, a, b;
    CBSPTree ** BSPTrees  = NULL;
    long        MeshCount = m_vpMeshList.size();
    CBounds3    BoundsA, BoundsB;
```

This function accepts no parameters and simply returns an HRESULT value used to report the success or failure of this process to the calling function.

At the start of this function we start by setting up a temporary ‘BSPOPTIONS’ structure. These will be the settings that we use to compile each of the mini-BSP trees built for every mesh undergoing the union operation. Within this structure we specify that we would like each tree to use the non-split compilation process. This will ensure that each tree stores an identical set of unsplit polygons as those contained in the original mesh. We also enable the ‘RemoveBackLeaves’ option. Recall that this setting causes the BSP compiler to forcibly remove any polygons that fall behind a terminal node. This effectively guarantees that a solid space indicator is inserted in place of each empty back leaf that might previously have been created. Should a mesh be slightly malformed, this will help to correct any problems that might arise due to the existence of illegal geometry. The remaining options are set to defaults that will provide a reasonable structure and a fast compile time.

```
    // Build the option set which will be used for all Mini-BSP Trees
    ZeroMemory( &Options, sizeof(BSPOPTIONS));
    Options.TreeType          = BSP_TYPE_NONSPLIT;
    Options.Enabled           = true;
    Options.RemoveBackLeaves  = true;
    Options.SplitHeuristic    = 3.0f;
    Options.SplitterSample    = 60;

    // Log – HSR: Building BSP Trees
```

With these options values initialized, we must now allocate the array in which each mesh’ BSP tree will be contained. This array of BSP tree pointers is stored in the local ‘BSPTrees’ variable declared at the top of the function. The mesh into which all the resulting polygon data can be placed must also be created at this point. Recall that this mesh object is stored within the ‘m_pResultMesh’ member variable.

```
    try
    {
        // Allocate memory for an array pf BSP Trees
        BSPTrees = new CBSPTree*[ MeshCount ];
        if (!BSPTrees) throw BCERR_OUTOFMEMORY;

        // Set all pointers to NULL
        ZeroMemory( BSPTrees, MeshCount * sizeof(CBSPTree*));

        // Allocate our result mesh
        m_pResultMesh = new CMesh;
        if (!m_pResultMesh) throw BCERR_OUTOFMEMORY;
```

The next step in this process is to compile a BSP tree for every mesh. The following block of code demonstrates how this is achieved.

For each mesh in the 'm_vpMeshList' array we allocate a new 'CBSPTree' object instance, and store its pointer in the newly allocated 'BSPTrees' container array. If the allocation was successful, the 'BSPOPTIONS' structure we set up at the start of this function is then passed into the new BSP tree object with a call to its 'SetOptions' method. Before we can compile the BSP tree, we must first pass in the polygon data from the current mesh object via the 'CBSPTree::AddFaces' method. With this process completed, we can finally call the BSP tree 'CompileTree' method in order for the mesh data to be compiled.

```
        // Build a BSP Tree for each mesh.
        for ( i = 0; i < MeshCount; i++ )
        {
            // Allocate a new tree
            BSPTrees[i] = new CBSPTree;
            if (!BSPTrees[i]) throw BCERR_OUTOFMEMORY;
            BSPTrees[i]->SetOptions( Options );

            // Add the faces from this mesh ready for compile
            hRet = BSPTrees[i]->AddFaces( m_vpMeshList[i]->Faces,
                                          m_vpMeshList[i]->FaceCount );
            if (FAILED(hRet)) throw hRet;

            // Compile the BSP Tree
            hRet = BSPTrees[i]->CompileTree();
            if (FAILED(hRet)) throw hRet;

            // Log – HSR : Update Progress

        } // Next Mesh
```

Now that the BSP tree data for each mesh has been compiled, the next task is to determine which of the mesh objects should be unioned together. While it is certainly possible for us to simply pass the polygon data of one mesh through the BSP tree of another, this could potentially cause hundreds of polygons to be split unnecessarily. This might be the case even if those two mesh objects were on opposite sides of the level. It is logical to assume therefore that if we can find any two meshes that intersect with one another then these two meshes should be unioned.

We search for these intersecting cases by first creating an outer loop that iterates through each of the meshes currently stored in the processor class. We will call this the source mesh. For each source mesh, we must then proceed to iterate through the mesh list once again in order to search for a another mesh that might intersect with the source. We should obviously ignore any cases in which both meshes are the same. However, it is also important to skip over any mesh that has already been processed – e.g. it was a source mesh at an earlier point and has already clipped and been clipped by every other mesh in the scene. Whenever a mesh has been fully processed within the outer loop, the BSP tree for this mesh will be released and set to NULL. If this is found to be the case in the inner loop, then we can safely ignore that mesh.

```
        // Log – HSR: Clipping Meshes for HSR
```

```
        // Now do the actual Union (HSR) clipping operations
        for ( a = 0; a < MeshCount; a++ )
        {
            // Log – HSR: Update progress

            // Skip any NULL bsp objects
            if (!BSPTrees[a] || BSPTrees[a]->GetFaceCount() == 0) continue;

            // Clip against all other meshes
            for ( b = 0; b < MeshCount; b++ )
            {
                // Skip any NULL bsp objects (i.e. has already been clipped)
                if (!BSPTrees[b] || BSPTrees[b]->GetFaceCount() == 0) continue;

                // Don't Boolean Op the mesh with itself
                if (a == b) continue;
```

At this stage we have a reference to two different meshes, neither of which has ever been clipped against the other. To get a rough idea of whether or not these two meshes intersect, we perform a quick broad-phase rejection step by testing for intersection between the bounding boxes of each mesh's BSP tree. If no intersection between these larger bounding boxes was found to have occurred, then we know that the meshes cannot possibly be intersecting. In this case we can continue on to test the next mesh in the inner loop.

If an intersection was found during this broad-phase test, then it is possible that the two meshes also intersect. To make sure that these two meshes really do intersect, we then perform a more accurate intersection test with a call to the 'CBSPTree::IntersectedByTree' function. We will cover this in detail shortly. For now all we need to know is that this test provides a more accurate result, using the BSP tree information constructed from the mesh data itself. Again, if this function indicated that there was no intersection, we can ignore this mesh combination and continue on to the next iteration of the inner loop.

```
                // Skip if the two don't intersect
                BoundsA = BSPTrees[a]->GetBounds();
                BoundsB = BSPTrees[b]->GetBounds();
                if ( !BoundsA.IntersectedByBounds( BoundsB,
                                                   CVector3( 0.1f, 0.1f, 0.1f )))
                    continue;

                if ( !BSPTrees[a]->IntersectedByTree( BSPTrees[b] ) )
                    continue;
```

If we reached this point in the function then it is safe to assume that these two meshes were likely to be intersecting and should be clipped. This is done in such a way that the portions of each mesh are removed if they are contained within the solid space of the other. We perform this clipping operation using the 'ClipTree' method of each BSP tree object. Notice however that the 'ClipTree' procedure accepts a pointer to another BSP tree as its first parameter. In fact we will not be touching any of the original mesh data during the HSR process because these meshes still belong to the compiler object. Instead we will be manipulating only the polygon data stored within each BSP tree object. Once the tree has been compiled, the data stored in its internal polygon array has no bearing on the actual structure of

the tree itself. As a result we can clip and manipulate this data as many times as we like without compromising the integrity of the tree.

If we look at the two calls made to the 'ClipTree' function, we can see that we first call this function on the tree created from the mesh selected in the outer loop, passing in the tree from the inner loop. In the next call this situation is reversed. In both cases we pass a value of 'true' to the second 'ClipSolid' parameter which forms the basis of the CSG union operation. Finally, pay special attention to the value passed to the final 'RemoveCoPlanar' parameter in each case. This will be important when we come to discuss the ClipTree function shortly.

If you have read through the textbook at this point, you should be aware that the CSG process can sometimes split polygons even if neither of the resulting fragments are ever deleted. For this reason we call each BSP tree objects' 'RepairSplits' method which is designed to undo these unnecessary splits. Again, we will examine this function a little later on.

```
                // Clip tree a to tree b and vice versa
                BSPTrees[a]->ClipTree( BSPTrees[b], true, false );
                BSPTrees[b]->ClipTree( BSPTrees[a], true, true  );

                // Repair Unrequired Splits
                BSPTrees[a]->RepairSplits();
                BSPTrees[b]->RepairSplits();

            } // Next Mesh b
```

With the BSP tree selected in the outer loop having now been clipped by each intersecting mesh in the scene, we can add the surviving polygons contained within this BSP tree object to the resulting mesh. This is achieved by calling the 'BuildFromBSPTree' we covered earlier. By specifying a value of false to the final parameter in this method, this will result in the polygon data being appended to those polygons already stored in the 'm_pResultMesh' object. Once the polygon data has been extracted from the tree, we then release it in order to free up any allocated resources that are no longer required. The applicable element in the 'BSPTrees' array is also finally set to a value of NULL. This is to ensure that we do not try and access the recently released BSP tree object at some point in the future.

```
            // Append all faces to the result brush
            m_pResultMesh->BuildFromBSPTree( BSPTrees[a], false );

            // Tree a has now been clipped by all other meshes
            delete BSPTrees[a];
            BSPTrees[a] = NULL;

        } // Next Mesh a

    } // End Try Block
```

The following catch block is designed to release any memory allocated by this method before returning the applicable error code to the calling function should an exception have been thrown at any point.

```
    catch ( HRESULT &e )
```

```
    {
        if ( BSPTrees )
        {
            for ( i = 0; i < MeshCount; i++ )
                if ( BSPTrees[i] ) delete BSPTrees[i];
            delete []BSPTrees;

        } // End if we allocated

        // Release the result mesh
        if ( m_pResultMesh ) { delete m_pResultMesh; m_pResultMesh = NULL; }

        // Log – HSR : Report Failure

        return e;

    } // End Catch Block
```

The final step in this process is simply to release any temporary objects and resources that remain allocated at the end of the process. Once this is complete, we finally return to the calling function with a code indicating success.

```
    // Release the BSP Trees
    if ( BSPTrees )
    {
        for ( i = 0; i < MeshCount; i++ ) if ( BSPTrees[i] ) delete BSPTrees[i];
        delete []BSPTrees;

    } // End if we allocated

    // Log – HSR : Report Success

    // Success!
    return BC_OK;

}
```

Once this entire process has been completed, the ‘m_pResultMesh’ mesh object will contain a copy of every polygon fragment that existed only within an empty area of space. With those fragments existing in solid space having been clipped away, this resulting mesh should now contain a valid set of scene polygon data with all hidden surfaces / illegal geometry removed.

CProcessHSR::GetResultMesh

As we know, the goal of the HSR processing module is to merge each of the specified meshes into one single result mesh. In the previous coverage of the ‘Process’ function defined by this class, we saw how the mesh referenced by the ‘m_pResultMesh’ member variable was constructed. This function is designed to be called by the compiler class in order to retrieve this mesh pointer ready for BSP compilation.

```
CMesh * CProcessHSR::GetResultMesh() const
```

```
{
    return m_pResultMesh;
}
```

CBSPTree – Additional CSG Functionality

With the general concepts involved in performing the hidden surface removal process behind us, let us take a look at that additional support functionality provided by the 'CBSPTree' class for performing CSG operations.

CBSPTree::ClipTree

The 'ClipTree' method of the BSP tree class provides the hidden surface removal processor with the functionality needed to perform the union operation used to merge each of the scene meshes together. Much like the 'BuildBSPTree' method discussed earlier in this chapter, this function is a recursive procedure that passes polygon data through the tree, classifying and splitting against each node until it reaches a terminal node within the hierarchy.

```
HRESULT CBSPTree::ClipTree( CBSPTree * pTree, bool ClipSolid, bool RemoveCoPlanar,
                            ULONG CurrentNode, CBSPFace * pFaceList )
{
    // 50 Bytes including Parameter list (based on __thiscall declaration)
    CBSPFace     *TestFace   = NULL, *NextFace  = NULL;
    CBSPFace     *FrontList  = NULL, *BackList  = NULL;
    CBSPFace     *FrontSplit = NULL, *BackSplit = NULL;
    unsigned long Plane      = 0;
    HRESULT       ErrCode    = BC_OK;

    // Validate Params
    if (!pTree) return BCERR_INVALIDPARAMS;

    // Did Someone use or pass in a silly tree ?
    if (pTree->GetFaceCount() < 1 || GetFaceCount() < 1 )
        return BCERR_BSP_INVALIDGEOMETRY;
```

Although there are 5 parameters declared by this function, only the first three should be passed to the initial call. The final two parameters are used only within this function to pass data between recursive calls. Let us briefly examine the purpose of each of these parameters and what should be passed to each of them.

*CBSPTree * pTree*

The first of these parameters is a pointer to another BSP tree object. The actual tree structure of the object passed to this parameter will *not* be used in this call. Only the polygon data stored within this tree will be manipulated. Recall that in order to perform any type of CSG operation on mesh data; each source mesh must have a BSP tree. With the tree also storing the mesh polygon information, it makes sense that we simply accept a pointer to the tree itself rather than the original mesh.

bool ClipSolid

This parameter is used by the calling function to control the circumstances under which clipped polygon fragments will be removed. If a value of ‘true’ is specified, then any source polygon fragments that end up in the solid space of this tree will be discarded. Conversely, if a value of ‘false’ is specified then those fragments ending up in *empty* space will be discarded. This parameter is the primary means by which we can adapt this same procedure for use with any of the CSG operations.

bool RemoveCoPlanar

This third parameter is used by the calling function to control the manner in which coplanar polygons are treated. If you have read the textbook at this point you should be familiar with the problems associated with the coplanar case in CSG operations. If you were performing the union of two meshes that had overlapping coplanar polygons, were you to remove both polygons you would be left with a hole. If however you were to keep both polygons you would introduced illegal geometry into the new mesh. By removing only one and keeping the other however, the overlapping fragment of the one mesh covers the gap left by the one removed. By alternating this value between the two calls needed to clip both meshes, this same behavior will be observed.

ULONG CurrentNode

This parameter contains the index of the current node in this BSP tree against which we are classifying the source polygon data. This parameter is optional and defaults to a value of 0 (the root node). It should not be specified in the initial call to this function.

*CBSPFace * pFaceList*

This fifth and final parameter is the head / root element in the linked list containing those polygons being clipped during this procedure. In much the same way as the construction of the BSP tree, the front and back lists collected at each step will be passed down to further recursions using this parameter. These polygons are actually references to those owned by the BSP tree object passed in as the first parameter. As with the ‘CurrentNode’, this parameter is optional and defaults to a value of NULL. It should not be specified in the initial call to this function.

With each of the parameters covered, let us move on to the primary implementation of this function.

The first real task with which this function is charged is to construct a linked list from each of the polygons contained in the BSP tree passed in as the first parameter. This is only performed if this is the first time in to this function (determined by the default NULL pointer contained with the pFaceList parameter) and will not take place in subsequent recursions. This is achieved simply by looping through each polygon in the tree using its ‘GetFaceCount’ and ‘GetFace’ accessor functions and attaching each polygon to the ‘Next’ pointer of the one before. Building a list of polygons from the source tree in this way allows us to work directly with its polygon data while maintaining the ability to reuse much of the functionality already designed to work with linked lists. Notice in the following code block that we also reset the ‘ChildSplit’ elements of each face to a default value of -1 in this same loop.

```
    // Automatically build lists
    if ( pFaceList == NULL )
```

```
    {
        // Build our sequential linked list
        for ( UINT i = 0; i < pTree->GetFaceCount(); i++ )
        {
            pTree->GetFace(i)->ChildSplit[0]= -1;
            pTree->GetFace(i)->ChildSplit[1]= -1;
            if ( i > 0 ) pTree->GetFace(i - 1)->Next = pTree->GetFace(i);

        } // Next Face

        // Reset last face just to be certain
        pTree->GetFace(i - 1)->Next = NULL;
        pFaceList = pTree->GetFace(0);

    } // End If No Faces
```

Now that we have access to either the linked list that has just been built in the first call to this function, or the linked list passed in subsequent recursive calls, we now begin to iterate through each polygon in this list. The first thing we must do in this loop is to make a backup of the pointer stored in that polygons 'Next' member. It is important that we do this here because the current polygon may be deleted or modified during this operation. It is also imperative that we ignore any polygons in the list that have been marked as deleted.

```
    // Select node plane and classify / send the list through the tree
    Plane = GetNode( CurrentNode )->Plane;

    for ( TestFace = pFaceList; TestFace; TestFace = NextFace )
    {
        // Store next face, as 'TestFace' may be modified / deleted
        NextFace = TestFace->Next;

        // Skip this polygon it has been deleted in some previous csg op
        if ( TestFace->Deleted ) continue;
```

The next step in the operation, much as in the BSP compile process, is to classify this polygon against the current node's plane and take any appropriate action based on the result.

The first case we deal with here is those polygons from the source BSP tree that are coplanar with the current node. If this polygon is found to be coplanar with the node and points in the same direction, then there is some additional logic that must be applied here to resolve the overlapping polygon problem we outlined earlier. Thankfully however, the solution is relatively simple. If the value passed to the 'RemoveCoPlanar' parameter is true then we simply attach this polygon to the back list being constructed in this call. If the value is false then we add it to the front list instead. Hopefully you can see why this is the case but in summary; should the polygon used to create the node in this tree overlap with the current polygon from the source tree, this source polygon would eventually end up in solid space and be clipped away. Conversely, passing it down the front would push it into an empty area and it would survive. In either case, should the polygon and the node face in opposite directions, the polygon should be added to the back list regardless.

```
        // Classify the polygon against the selected plane
        switch ( GetPlane(Plane)->ClassifyPoly( TestFace->Vertices,
```

```
                                                TestFace->VertexCount,
                                                sizeof(CVertex) ) )
        {
            case CLASSIFY_ONPLANE:
                // Test the direction of the face against the plane.
                if ( GetPlane(Plane)->SameFacing( TestFace->Normal ) )
                {
                    if ( RemoveCoPlanar )
                    {
                        TestFace->Next = BackList;
                        BackList       = TestFace;
                    }
                    else
                    {
                        TestFace->Next = FrontList;
                        FrontList = TestFace;
                    }
                }
                else
                {
                    TestFace->Next = BackList;
                    BackList       = TestFace;

                } // End if Plane Facing
                break;
```

If the polygon fragment was classified as being contained completely within the front halfspace of the node, we simply add it to the current front list in this case.

```
            case CLASSIFY_INFRONT:
                // Pass the face straight down the front list.
                TestFace->Next     = FrontList;
                FrontList          = TestFace;
                break;
```

In a similar fashion as the in-front case, should the polygon fragment be completely behind the node then it is added straight to the back list.

```
            case CLASSIFY_BEHIND:
                // Pass the face straight down the back list.
                TestFace->Next     = BackList;
                BackList           = TestFace;
                break;
```

The initial stages of the spanning case are almost identical to that of the ‘BuildBSPTree’ function. The only real difference here is that the split polygon fragments that we generate are added to the *source* BSP tree passed in to the first parameter. Notice how we call the ‘IncreaseFaceCount’ and ‘SetFace’ methods of the ‘pTree’ object, rather than those local to this object.

```
            case CLASSIFY_SPANNING:
                // Allocate new front within the passed tree
                if (!(FrontSplit = AllocBSPFace()))
```

```
                {
                    ErrCode = BCERR_OUTOFMEMORY;
                    goto ClipError;
                }
                if (!pTree->IncreaseFaceCount())
                {
                    ErrCode = BCERR_OUTOFMEMORY;
                    goto ClipError;
                }

                pTree->SetFace( pTree->GetFaceCount() - 1, FrontSplit );

                // Allocate new back fragment within the passed tree
                if (!(BackSplit     = AllocBSPFace()))
                {
                    ErrCode = BCERR_OUTOFMEMORY;
                    goto ClipError;
                }
                if (!pTree->IncreaseFaceCount())
                {
                    ErrCode = BCERR_OUTOFMEMORY;
                    goto ClipError;
                }
                pTree->SetFace( pTree->GetFaceCount() - 1, BackSplit );
```

With these two new polygon objects created, we can now proceed to split the polygon against the node plane, generating the two new polygon fragments from those portions of the original source polygon that lay on either side.

In the spanning case of the BSP compiler, we follow the split operation by deleting the original polygon. During the process of clipping polygons for CSG however, there are cases in which the splitting of polygon data can occur even when neither fragment will end up being deleted. As a result, it is possible to repair the unnecessary splits but only if the original polygon remains in-tact and accessible. For this reason we simply set the polygon's 'Deleted' member to true in this function. This will still have the effect of causing this polygon to be ignored by this function due to the initial check above. The last piece of information we need to update in the original polygon are the 'ChildSplit elements. Into these we place the indices to the two new polygons as they exist in the **source** BSP tree object. This information will be used during the 'RepairSplits' method we will discuss shortly. Finally we add each of the two new fragments to the relevant front or back list ready for future processing.

```
                // Split the polygon
                if (FAILED( ErrCode = TestFace->Split(*GetPlane(Plane),
                                                      FrontSplit,
                                                      BackSplit))) goto ClipError;

                // Just flag as deleted and mark for potential repair
                TestFace->Deleted    = true;
                TestFace->ChildSplit[0] = pTree->GetFaceCount() - 2;
                TestFace->ChildSplit[1] = pTree->GetFaceCount() - 1;

                // Add it to the head of our recursion lists
                FrontSplit->Next = FrontList;
                FrontList        = FrontSplit;
```

```
                BackSplit->Next  = BackList;
                BackList         = BackSplit;

                break;

        } // End Switch

    } // Next Face
```

With every polygon in the face list now processed, we should have fully populated the ‘FrontList’ and ‘BackList’ with all relevant data. At this point in the function we begin the actual process of removing any applicable polygons should we have arrived at a leaf.

We begin with those tests performed when the ‘ClipSolid’ parameter was set to a value of ‘true’. As we know, the only place that solid space can exist within our polygon-aligned BSP leaf tree is behind a node. As a result we test the value of the ‘Back’ member of the current node to see if it is equal to the ‘BSP_SOLID_LEAF’ constant value. If this is the case then we know that the space behind this leaf is solid and therefore the polygons contained within the back list can be deleted. Remember that we must not physically release the polygon data at any point in this clipping process; we simply set each polygon’s ‘Deleted’ member to a value of ‘true’. Once we have finished iterating through each of the polygons in the list, we must finally clear the ‘BackList’ variable by overwriting it with a value of NULL. This should be done in order to prevent this list of now deleted polygons from being passed down the back of the node later in this function.

```
    // Now onto the clipping
    if ( ClipSolid )
    {
        if ( GetNode(CurrentNode)->Back == BSP_SOLID_LEAF )
        {
            // Iterate through and flag all back polys as deleted
            for ( TestFace = BackList; TestFace; TestFace = TestFace->Next )
                TestFace->Deleted = true;

            // Empty Back List
            BackList = NULL;

        } // End if Back == Solid

    }  // End if clipping solid
```

If the value passed to the ‘ClipSolid’ parameter is equal to ‘false’, then we have been instructed to remove those polygon fragments that may have ended up in any *empty* leaf. Unlike the solid leaf case, if the ‘RemoveBackLeaf’ BSP compilation option has been disabled then empty leaves can exist both in front of and behind any node. For this reason we must test both the ‘Front’ and ‘Back’ members of the current node in this case. If an empty leaf is found to be attached to the back of this node, then again each of the polygons in the back list should be marked as deleted. Conversely, if an empty leaf is found to be attached to the front of this node, then each of the polygons in the *front* list should be marked as deleted. In either case, to prevent any further recursion with this data, we must also remember to clear the applicable ‘FrontList’ or ‘BackList’ variable once the polygons have been processed as before.

```
    else
    {
        if ( GetNode(CurrentNode)->Back < 0 )
        {
            // Iterate through and flag all back polys as deleted
            for ( TestFace = BackList; TestFace; TestFace = TestFace->Next )
                TestFace->Deleted = true;

            // Empty Back List
            BackList = NULL;

        } // End if Back == Empty

        if ( GetNode(CurrentNode)->Front < 0 )
        {
            // Iterate through and flag all front polys as deleted
            for ( TestFace = FrontList; TestFace; TestFace = TestFace->Next )
                TestFace->Deleted = true;

            // Empty Front List
            FrontList = NULL;

        } // End if Front == Empty

    } // End if clipping empty
```

It is not necessarily the case that we would have encountered leaves at this stage in the recursive process. As a result, our front and back lists may still be fully populated. Due to the fact that we are actually passing data through the tree in order to determine the leaves in which they are contained, it is necessary for us to traverse into any front and back child nodes at this point. In each case we pass in the same parameter information as was passed to this call with the exception of the last two parameters. Into these we pass the relevant front & back node indices, along with the matching ‘FrontList’ or ‘BackList’ variable. Once each side of the node have been fully traversed, we finally return from this function and pass control back to the caller.

```
    // Pass down the front of the node
    if ( FrontList && GetNode(CurrentNode)->Front >= 0 )
            ErrCode = ClipTree( pTree, ClipSolid, RemoveCoPlanar,
                                GetNode(CurrentNode)->Front, FrontList);

    if ( FAILED( ErrCode ) ) goto ClipError;

    // Pass down the back of the node
    if ( BackList && GetNode(CurrentNode)->Back >= 0 )
            ErrCode = ClipTree( pTree, ClipSolid, RemoveCoPlanar,
                                GetNode(CurrentNode)->Back, BackList);

    if ( FAILED( ErrCode ) ) goto ClipError;

    // Success!
    return BC_OK;
```

Because this function is not responsible for creating or deleting any polygon objects that exist only within the stack based linked lists, we do not need to perform any kind of garbage collection in the error handling case. Instead, whenever this function skips to this piece of code we simply return the current error code maintained within the 'ErrCode' variable.

```
ClipError:
    // Failed
    return ErrCode;
}
```

Once this function has completed its recursive execution, any polygons that were contained within the source BSP tree passed to the first parameter of this function will now contain a list of all the newly clipped fragments, in addition to the original polygons simply marked as 'Deleted'. However, no modification of the object in which the 'ClipTree' method was called has occurred. For this tree to subsequently be clipped, the HSR processing module would now perform the same operation again but with the trees reversed. This tree should now be passed in to the first parameter of the 'ClipTree' method of the other.

CBSPTree::RepairSplits

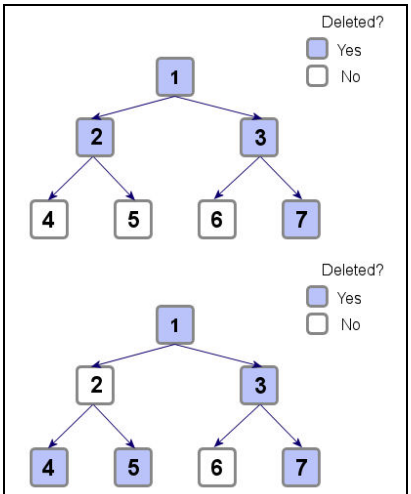

Figure 16.5

The 'RepairSplits' function is an extremely small and yet very fast and elegant way of repairing any unnecessary splits that may have been introduced during the constructive solid geometry clipping process. During the 'ClipTree' process, recall that we store the indices to those polygons created whenever a polygon was split. In addition to this, the original polygon is simply marked as deleted rather than being physically removed.

Referring to the uppermost diagram in figure 16.5, we demonstrate a situation in which the original polygon – shown with an index of 1 – has been split into two child fragments – 2 & 3. This original polygon has then been marked as deleted. Later in the process, each of these child fragments has been split into a further two. Again the source polygons have been flagged as deleted. Finally, polygon 7 gets flagged as deleted during the clipping process. At this point we have a situation in which there are only three polygon fragments that are not marked as deleted. These are the polygons 4, 5 and 6. Notice that polygons 4 and 5 both originated from the same parent – polygon 2. This clearly demonstrates an unnecessary split because both child fragments have survived.

If we now examine the bottom diagram in this same figure we can see what might happen if we repair the unnecessary splits in this scenario. We can see that the child fragments 4 and 5 have now been marked as deleted, and their parent polygon 2 has been resurrected. Its sibling polygon 3 however remains in the deleted state because only one of its child fragments survived.

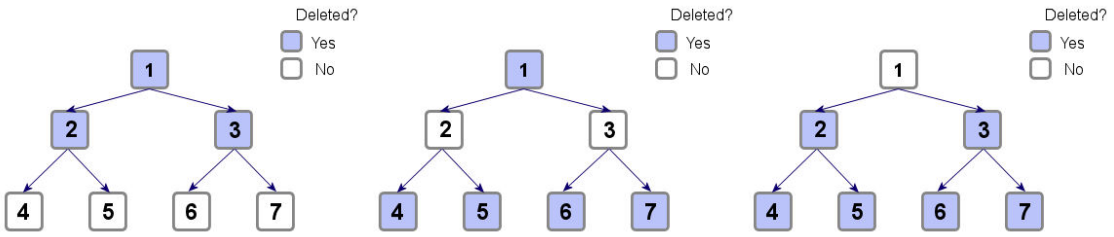

Figure 16.6

Figure 16.6 shows a series of diagrams that demonstrate an alternative scenario. In this case, all 4 of the bottom most children have survived. Both splits generated against polygons 2 and 3 were clearly therefore unnecessary and can be repaired. At this stage, the four child fragments have been marked as deleted and the original fragments 2 & 3 have been restored. Again in this case, these two polygons are fragments resulting from the original polygon 1 being split. Since both child fragments are no longer deleted we can go one step further and restore the original polygon.

As you might imagine, this simple repair process could greatly reduce the number of polygons that might result from an extensive, high-resolution clipping operation. With the theories discussed, let us see how this process is implemented in our 'CBSPTree' class.

The first thing we must do in this function is set up a loop that starts with the last polygon in the BSP tree's polygon array, and works **backwards** to the beginning. If you think back to figure 16.6 remember that we were able to perform multiple levels of repair when starting at the four child fragments and working our way up. In the BSP tree's 'ClipTree' function, whenever new polygon fragments are generated they are added to the *end* of the tree's polygon array with the earliest originals at the start.

Once inside this loop, we first test the contents of one of the 'ChildSplit' elements of the current polygon to see if it makes reference to any split fragment. If either of these elements contains a value other than -1 then this means that this fragment was split at some point. Once we know that a polygon was split into two child fragments, we can then retrieve those fragments and test their 'Deleted' member value. If *neither* fragment is marked as deleted then we will be able to repair this fragment. As a result we set the current polygon's 'Deleted' member to false, and each of the child fragments 'Deleted' members to true.

```
void CBSPTree::RepairSplits()
{
    // For each face
    for ( int i = GetFaceCount() - 1; i >= 0; i--)
    {
        // If one isn't -1, neither is the other
        if ( GetFace(i)->ChildSplit[0] != -1)
        {
            // If the two children are valid this split should be repaired
            if ( !GetFace( GetFace(i)->ChildSplit[0] )->Deleted &&
                 !GetFace( GetFace(i)->ChildSplit[1] )->Deleted )
            {
                // Restore the parent, and delete the children
                GetFace( i )->Deleted = false;
```

```
                GetFace( GetFace(i)->ChildSplit[0] )->Deleted = true;
                GetFace( GetFace(i)->ChildSplit[1] )->Deleted = true;

            } // End if Both Valid

        } // End if Has Children

    } // Next Face
}
```

With this polygon repaired, we continue working backward through the array. Eventually we might come to an earlier polygon that references the polygon we had just repaired. This might be deleted once again, along with a sibling, have *its* parent restored. Once this entire process has completed, we should find that in most cases this repair process is extremely effective because of the fact that it is able to perform partial repairs on original polygons.

CBSPTree::IntersectedByTree / IntersectedByFace

The 'IntersectedByTree' method is the first of two companion functions outlined here. This function is designed to be called by the application in order to test for intersection between two BSP tree objects. The first is the object into which the application is calling, and the second is passed in as the single parameter to this function. This is achieved by passing each polygon contained within the specified BSP tree, through the local tree structure to determine if any of those polygons fall into solid space.

This function is however just a wrapper for the second of these two functions – the 'IntersectedByFace' method. Iterating through each polygon in the BSP tree object passed, this function passes that polygon directly into that method and immediately returning a value of true should it report an intersection.

```
bool CBSPTree::IntersectedByTree( const CBSPTree * pTree ) const
{
    // Loop through each face testing for intersection
    for ( ULONG i = 0; i < pTree->GetFaceCount(); i++ )
    {
        if ( IntersectedByFace( pTree->GetFace(i) ) ) return true;

    } // Next Face

    // No Intersection
    return false;
}
```

The 'IntersectedByFace' method is a recursive procedure and serves as the core of this intersection testing process. This function accepts two parameters. The first is a pointer to the single polygon object to be tested and the second is the index to the current node at this level in the hierarchy. The premise of this function is to pass the specified polygon through the tree, and return an 'intersection' result if that polygon was found to exist in solid space.

> **Note:** This intersection test is not 100% accurate and is designed this way for reasons of efficiency. In order to accurately detect when a polygon falls into solid space it should be split against any node plane it was found to be spanning. Because we do not split the polygon in this way, it is possible for a false

intersection to be returned in certain cases. However, this technique should never return a non-intersection result in cases where an intersection definitely occurred.

```
bool CBSPTree::IntersectedByFace( const CFace * pFace, ULONG Node /* = 0 */ ) const
{
    // Validate Params
    if (!pFace || Node < 0 || Node >= GetNodeCount() ) return false;

    int NodeFront = GetNode( Node )->Front;
    int NodeBack  = GetNode( Node )->Back;
```

The first thing we do at the top of this function is to retrieve the values contained in the front and back members of the current node. This is done for convenience purposes so that we don't have to retrieve them each time we need them.

The next step in the operation is to classify this polygon against the current node's plane and take any appropriate action based on the result.

The first case we deal with here occurs when the polygon we are testing is either co-planar with the node plane, or is found to be spanning. It might seem a little strange to combine both the onplane and spanning case as we have below, but in this function we will not be splitting the test polygon. Instead, all we are interested in is whether there is any part of the polygon in front, behind or on both sides of the node plane. If the polygon is found to be spanning or coplanar then it will be considered to exist both in front and behind. As a result, we check the back of the node to see if it describes solid space. If it does then we immediately return true because we have found a potential intersection. If there is no solid leaf behind, we simply pass this polygon first down the front of the current node and then down the back should a child node exist there in each case. If the recursive call to either 'IntersectedByFace' function returns true, we must also return true such that we will step all the way back out of the recursive procedure.

```
    // Classify this poly against the nodes plane
    switch ( GetPlane(GetNode(Node)->Plane)->ClassifyPoly( pFace->Vertices,
                                                           pFace->VertexCount,
                                                           sizeof(CVertex) ) )
    {
        case CLASSIFY_SPANNING:
        case CLASSIFY_ONPLANE:
            // Solid Leaf
            if (NodeBack == BSP_SOLID_LEAF ) return true;
            // Pass down the front
            if (NodeFront >= 0 )
            {
                if ( IntersectedByFace( pFace, NodeFront ) ) return true;
            }
            // Pass down the back
            if (NodeBack  >= 0 )
            {
                if ( IntersectedByFace( pFace, NodeBack  ) ) return true;
            }
            break;
```

The next case is that in which the polygon is contained completely in the front half space of the current node. Since no solid leaf can exist in front of a node, we simply pass the current polygon down the front of this node should another child node exist there. Again, we return true immediately if this recursive informed us of an intersection.

```
        case CLASSIFY_INFRONT:
            // Pass down the front
            if (NodeFront >= 0 )
            {
                if ( IntersectedByFace( pFace, NodeFront ) ) return true;
            }
            break;
```

Similar to the front case, this case occurs when the polygon is completely contained in the back half space of the node. Solid space can exist *behind* a node however, so we test for this and return true if solid space is found to exist there. If not, we pass the polygon down the back of the current node should another child node exist there.

```
        case CLASSIFY_BEHIND:
            // Solid Leaf
            if (NodeBack == BSP_SOLID_LEAF ) return true;
            // Pass down the back
            if (NodeBack  >= 0 )
            {
                if ( IntersectedByFace( pFace, NodeBack  ) ) return true;
            }
            break;

    } // End Classify Switch
```

If we get here then we did not find any intersections at this level in the tree / recursion. As a result we simply return false from this function in this case.

```
    // No Intersection at this level
    return false;
}
```

This function concludes our coverage of the additional CSG support functionality provided by the ‘CBSPTree’ class. Being the only module that remains to be covered, let us now examine the T-Junction repair process module.

The CProcessTJR Module Class

The T-Junction repair process is the last of the three compiler modules we will implement in this lab project. It is thankfully also the most simple. Because the majority of the code within this process is taken directly from our previous implementation of the T-Junction repair process, we will not dwell on this topic for too long. Before we move on to the modifications that we have made to the procedures, let us take a look at the module class declaration.

```
class CProcessTJR
{
public:
    // Constructors & Destructors for This Class.
             CProcessTJR();
    virtual ~CProcessTJR();

    // Public Member Functions Omitted

private:
    // Private Variables for This Class.
    TJROPTIONS      m_OptionSet;        // The TJR option set
    ILogger        *m_pLogger;          // Log output interface.
    CCompiler      *m_pParent;          // Parent Compiler Pointer

    // Private Member Functions Omitted

};
```

The declaration for the CProcessTJR module class is very similar to that of the hidden surface removal class we looked at earlier. Due to the fact that this module works directly on the existing scene data however, there are even fewer member variables.

Although the member variables declared within this class are only those required by every processing module, let us quickly examine how they apply to this class.

TJROPTIONS m_OptionSet
This member stores the various settings, specified by the application to inform the compiler how the T-Junction repair operation should be executed. Although this structure contains no settings outside of the 'Enabled' Boolean, this options structure should be set using the 'SetOptions' accessor function in case of future enhancement.

ILogger * m_pLogger
In order for the TJR processor to report progress, error and useful status information to the user, this member variable stores a pointer to the application defined logging class. While the logging feature remains optional, this member can be set with a call to the 'SetLogger' accessor function defined by this class. In this lab project, the logging class instance is passed to this object by the 'CCompiler::PerformJTR' function.

CCompiler * m_pParent
Each compilation module class maintains a member that stores a reference to the parent compiler class that created it. This member is set with a call to the 'SetParent' accessor function defined by each of these classes. This information is used by each of these processing modules in order to test the current state of the compiler to determine if the compile operation has been either paused or cancelled. Although we have omitted much of the logging and progress functionality from the code listings in this workbook, you should take a look at the source code for lab project 16.2 to see this process in action.

TJR Module Accessor Functions

There are only a few accessor functions defined within the TJR processing module class. In each case, the body for these functions can be found in the class declaration contained in the header file, rather than in the source module / .CPP file. We will not go into detail about how each of these functions is implemented, for this you should take a look at the 'CProcessTJR.h' header file contained in the 'Compiler Source' project directory.

The only accessor functions defined by the 'CProcessTJR' class are those that set the various pieces of information required by each compilation process module. These include the process options, a pointer to the parent compiler and a pointer to the class defined logging class. These functions simply store those specified values into the applicable member variables declared by this class.

```
void            SetOptions    ( TJROPTIONS Options    )
void            SetParent     ( CCompiler * pCompiler )
void            SetLogger     ( ILogger * pLogger     )
```

Now that we are familiar with each of this class's member variables, let us move on to examine the main process functionality.

CProcessTJR::Process

This function is called by the compiler's 'PerformTJR' function in order to begin the actual T-Junction repair process. This process has no pre-requisites in the form of data being added to the module class in advance. Instead, this method will work directly with that scene data passed into its parameters.

Earlier we discussed the fact that this process is not concerned with any of the renderable polygon information such as textures or materials, or even the surface normal. All we need to have access to are the vertices stored within the scene polygons. As a result, the first parameter declared by this function accepts a list of *pointers* to one or more base 'CPolygon' objects. The second parameter specifies the number of polygons contained in that array.

```
HRESULT CProcessTJR::Process( CPolygon ** ppPolys, ULONG PolyCount )
{
    ULONG      i, k;
    CBounds3  *pBounds = NULL;
    CPolygon  *pCurrentPoly, *pTestPoly;

    // Validate values
    if (!ppPolys || !PolyCount) return BCERR_INVALIDPARAMS;
```

In our previous implementation, we took advantage of the compiled tree in order to rapidly retrieve a list of neighbors for any polygon currently being tested. In this case however we cannot guarantee that a BPS tree has even been compiled. As a result, this function must make its own adjacency arrangements. In order to provide a relatively quick means by which we can determine these neighboring polygons, we first build a list of bounding boxes for every polygon passed into this function using the bounding box's 'CalculateFromPoly' function. We also increase the size of each bounding box by a small amount. This is in order to ensure that the neighbor polygon bounding boxes safely overlap.

```
    try
```

```
    {

        // Allocate space for bounding boxes
        if (!(pBounds = new CBounds3[PolyCount])) return BCERR_OUTOFMEMORY;

        // Log – TJR: Pre-compiling adjacency information

        // Calculate polygon bounds
        for ( i = 0; i < PolyCount; i++ )
        {
            // Log – TJR: Update Progress

            // Build polygon bounds
            pBounds[i].CalculateFromPolygon( ppPolys[ i ]->Vertices,
                                             ppPolys[ i ]->VertexCount,
                                             sizeof(CVertex) );

            // Increase bounds slightly for tolerance
            pBounds[i].Min.x -= 0.1f;
            pBounds[i].Min.y -= 0.1f;
            pBounds[i].Min.z -= 0.1f;
            pBounds[i].Max.x += 0.1f;
            pBounds[i].Max.y += 0.1f;
            pBounds[i].Max.z += 0.1f;

        } // Next Bounds
```

With this bounding box information constructed, we can begin the process of searching for two polygons whose bounding boxes intersect. In a similar manner to the hidden surface removal process, we construct two loops here. The first iterates through each polygon in the outer loop, and another iterates through each polygon in the inner loop. Assuming these aren't both the same polygon, we test for intersection between both polygon bounding boxes with a call to the 'CBounds3::IntersectByBounds' method.

```
        // Log – TJR: Repairing T-Junctions

        // Loop through Faces
        for ( i = 0; i < PolyCount; i++ )
        {
            // Log – TJR: Update Progress

            // Get the current poly to test and its vertex array
            pCurrentPoly = ppPolys[ i ];
            if (!pCurrentPoly) continue;

            // Test against every other face in the tree
            for ( k = 0; k < PolyCount; k++ )
            {
                // Don't against test self
                if (i == k) continue;

                // Get the test face and its vertices
                pTestPoly = ppPolys[ k ];
                if (!pTestPoly) continue;
```

```
                // If the two do not intersect then there is no need for testing.
                if ( !pBounds[i].IntersectedByBounds( pBounds[k] ) ) continue;
```

If the two bounding boxes did intersect at this point then we attempt to run the repair process between each of these polygons, first in one direction and then in the other. We then move on to the next polygon in the inner loop.

```
                // Repair against the testing poly
                if (!(RepairTJunctions( pCurrentPoly, pTestPoly )))
                    throw BCERR_OUTOFMEMORY;

                // Now we do exactly the same but in reverse order
                if (!(RepairTJunctions( pTestPoly, pCurrentPoly )))
                    throw BCERR_OUTOFMEMORY;

            } // Next Test Face
```

At this stage, the current polygon in the outer loop has been processed and repaired against every other polygon in the scene and it need not be considered again in the future. As a result, we overwrite its element in the array with a NULL pointer causing the each loop to bypass this element.

```
            // Completely processed
             ppPolys[i] = NULL;

        } // Next Current Face

    } // End Try Block
```

The final part of this function simply implements the exception handling and clean up code common to most of the functions in this application. Assuming no exception is thrown, the T-Junction repair process has now completed successfully and can return a success code to the calling function.

```
    catch ( HRESULT &e )
    {
        // Clean up and return (failure)
        if (pBounds) delete []pBounds;

        // Log – TJR: Report Failure

        return e;

    } // End Catch Block

    // Release used memory
    if (pBounds) delete []pBounds;

    // Log – TJR: Report Success

    // Success!
    return BC_OK;

}
```

Although this wrapper function contains a reasonable amount of logic on its own, the actual process of repairing the T-Junctions is implemented in the ‘RepairTJunctions’ method covered in earlier lessons.

CProcessTJR::RepairTJunctions

The ‘RepairTJunctions’ method outlined below is identical to that we have previously covered and implemented. This is with the exception of it having been adapted for use with the new math support classes. As a result of these minor alterations, we have included this code again here for your convenience.

```
bool CProcessTJR::RepairTJunctions( CPolygon *pPoly1, CPolygon *pPoly2 ) const
{
    CVector3   Delta;
    float      Percent;
    ULONG      v1, v2, v1a;
    CVertex    Vert1, Vert2, Vert1a;


    // Validate Parameters
    if (!pPoly1 || !pPoly2) return false;

    // For each edge of this face
    for ( v1 = 0; v1 < pPoly1->VertexCount; v1++ )
    {
        // Retrieve the next edge vertex (wraps to 0)
        v2 = ((v1 + 1) % pPoly1->VertexCount);

        // Store verts (Required because indices may change)
        Vert1 = pPoly1->Vertices[v1];
        Vert2 = pPoly1->Vertices[v2];

        // Now loop through each vertex in the test face
        for ( v1a = 0; v1a < pPoly2->VertexCount; v1a++ )
        {
            // Store test point for easy access
            Vert1a = pPoly2->Vertices[v1a];

            // Test if this vertex is close to the test edge
            // (Also returns out of range value if the point is past the line ends)
            if ( Vert1a.DistanceToLine( Vert1, Vert2 ) < EPSILON )
            {
                // Insert a new vertex within this edge
                long NewVert = pPoly1->InsertVertex( v2 );
                if (NewVert < 0) return false;

                // Set the vertex pos
                CVertex * pNewVert = &pPoly1->Vertices[ NewVert ];
                pNewVert->x = Vert1a.x;
                pNewVert->y = Vert1a.y;
                pNewVert->z = Vert1a.z;

                // Calculate the percentage for interpolation calcs
                Percent = (*pNewVert - Vert1).Length() / (Vert2 - Vert1).Length();
```

```
                // Interpolate texture coordinates
                Delta.x      = Vert2.tu - Vert1.tu;
                Delta.y      = Vert2.tv - Vert1.tv;
                pNewVert->tu = Vert1.tu + ( Delta.x * Percent );
                pNewVert->tv = Vert1.tv + ( Delta.y * Percent );

                // Interpolate normal
                Delta            = Vert2.Normal - Vert1.Normal;
                pNewVert->Normal = Vert1.Normal + (Delta * Percent);
                pNewVert->Normal.Normalize();

                // Update the edge for which we are testing
                Vert2 = *pNewVert;

            } // End if on edge

        } // Next Vertex v1a

    } // Next Vertex  v1

    // Success!
    return true;
}
```

Lab Project Conclusion

This has been a very interesting but difficult lab project to cover. With the introduction of a completely new framework came new and interesting challenges to overcome. With that said, we now have at our disposal a strong foundation application that can be greatly enhanced as you progress further in your studies with us. With the introduction of the polygon-aligned leaf BSP tree and hidden surface removal processes too, we now have some very exciting opportunities opening up to us. In the next lesson we will be taking these concepts even further with the introduction of portals for leaf connectivity information, and PVS for occlusion culling and visibility which is certainly a topic to look forward to.